

第二版

使用Perl实现系统管理自动化

David N. Blank-Edelman 著

盛 春 蒋永清 王 晖 译

O'REILLY®

Beijing • Cambridge • Farnham • Köln • Sebastopol • Tokyo

O'Reilly Media, Inc. 授权东南大学出版社出版

东南大学出版社

图书在版编目 (CIP) 数据

使用 Perl 实现系统管理自动化: 第 2 版: 中文 / (美)
布兰克 - 艾德尔曼 (Blank-Edelman, D.N.) 著; 盛春,
蒋永清, 王晖译. —南京: 东南大学出版社, 2011.12

书名原文: Automating System Administration with Perl,
Second Edition

ISBN 978-7-5641-3072-5

I. ①使… II. ①布… ②盛… ③蒋… ④王… III. ① Perl
语言—程序设计 IV. ① TP312

中国版本图书馆 CIP 数据核字 (2011) 第 227001 号

江苏省版权局著作权合同登记

图字: 10-2010-277 号

©2009 by O'Reilly Media, Inc.

Simplified Chinese Edition, jointly published by O'Reilly Media, Inc. and Southeast University Press,
2011. Authorized translation of the English edition, 2009 O'Reilly Media, Inc., the owner of all rights to
publish and sell the same.

All rights reserved including the rights of reproduction in whole or in part in any form.

英文原版由 O'Reilly Media, Inc. 出版 2009。

简体中文版由东南大学出版社出版 2011。英文原版的翻译得到 O'Reilly Media, Inc. 的授权。此简体中文版的出版和销售得到出版权和销售权的所有者——O'Reilly Media, Inc. 的许可。

版权所有, 未得书面许可, 本书的任何部分和全部不得以任何形式重制。

使用 Perl 实现系统管理自动化 第二版 (中文版)

出版发行: 东南大学出版社

地 址: 南京四牌楼 2 号 邮编: 210096

出 版 人: 江建中

网 址: <http://www.seupress.com>

电子邮件: press@seupress.com

印 刷: 扬中市印刷有限公司

开 本: 787 毫米 × 980 毫米 16 开本

印 张: 39 印张

字 数: 764 千字

版 次: 2011 年 12 月第 1 版

印 次: 2011 年 12 月第 1 次印刷

书 号: ISBN 978-7-5641-3072-5

印 数: 1~3000 册

定 价: 94.00 元 (册)

本社图书若有印装质量问题, 请直接与读者服务部联系。电话 (传真): 025-83792328

致 Cindy，你是我一生的最爱。致 Elijah，给你诚挚的祝福。

目录

前言	1
第 1 章 简介	9
自动化是必须的	9
Perl如何帮助你.....	10
这本书会带你学会.....	11
你需要什么	12
关于本书使用的Perl版本的解释	13
怎么不用Perl 5.10?	13
那怎么没介绍Strawberry Perl?	13
那么Perl 6呢?	14
如何在Windows Vista中使用范例代码.....	14
载入和使用模块	15
在Unix上安装模块	16
在Win32上安装模块.....	17
要成为万能的并不容易	17
尽量避免提升权限	17
尽可能早地放弃特权.....	18
小心读取数据.....	18
小心写数据.....	19

避免竞争条件	20
保持乐观	20
更多参考资料	20
第2章 文件系统	22
Perl用于拯救数据	22
文件系统差异	23
Unix	23
基于Windows的操作系统	23
Mac OS X	25
文件系统差异汇总	25
使用Perl处理文件系统差异	26
手动遍历文件系统	28
使用File::Find模块来遍历文件系统	33
使用File::Find::Rule模块来遍历文件系统	42
操纵磁盘限额	44
通过edquota技巧来编辑磁盘限额	46
使用 Quota 模块来编辑限额	50
在Windows下编辑NTFS限额	51
查询文件系统使用量	52
本章所用模块	53
更多参考资料	54
第3章 用户账户	55
Unix用户身份	56
经典Unix密码文件	56
BSD 4.4对密码文件的改动	62
影子密码	63
基于Windows的操作系统用户身份	64
Windows用户信息存储和访问	64
Windows用户ID编号	66
Windows密码和Unix密码不兼容	68

Windows组	68
Windows 用户权力	72
构建用户账户管理系统	76
后端数据库	78
底层组件库	82
处理脚本	93
账户系统总结	98
本章所用模块	100
更多参考资料	101
Unix密码文件	101
Windows用户管理	101
第4章 用户活动	103
进程管理	104
基于Windows的操作系统进程控制	104
Unix进程控制	122
文件及网络操作	129
在Windows上跟踪文件操作	129
在Windows上跟踪网络操作	132
在Unix上跟踪文件和网络操作	134
本章所用模块	140
安装 Win32::Setupsup	140
更多参考资料	140
第5章 TCP/IP名称和配置服务	142
Host文件	142
生成host文件	145
在host文件生成过程中的错误检查	148
改善host文件输出	149
引入源代码控制系统	152
NIS、NIS+和WINS	156
NIS+	158

Windows Internet名称服务 (WINS).....	159
域名服务 (DNS)	159
生成DNS (BIND) 配置文件.....	160
DNS检查：迭代方式.....	169
DHCP	178
主动探测不良DHCP服务器.....	180
监控正规DHCP服务器	184
本章所用模块	187
更多参考资料	187
第6章 使用配置文件工作	188
配置文件格式	190
二进制格式.....	191
使用分隔符的文本格式.....	192
键-值对格式	192
置标语言	195
多功能合一模块	238
高级配置信息存储机制	239
本章所用模块	239
更多参考资料	240
XML和YAML.....	240
第7章 SQL数据库管理	242
从Perl中与SQL服务器交互	243
使用DBI框架	245
从DBI中调用ODBC	251
服务器文档化	253
通过DBI访问MySQL服务器	254
通过DBI访问Oracle服务器	256
通过ODBC访问Microsoft SQL Server.....	257
登录数据库.....	260
监控数据库服务器上的空间使用	262
本章所用模块	265

更多参考信息	265
DBI.....	265
Microsoft SQL Server	266
ODBC.....	266
Oracle.....	266
第8章 E-mail	267
发送邮件	267
获取sendmail（或其他类似的邮件传输代理）	268
使用特定操作系统的IPC框架驱动邮件客户端	268
直接使用邮件协议发送	270
发送邮件时的常见错误	275
持续不断发送邮件	275
无用主题行.....	284
消息正文中信息不足.....	284
收取邮件	287
使用POP3收取邮件.....	287
使用IMAP4rev1收取邮件.....	288
处理邮件	293
剖析单一邮件.....	293
剖析整个邮箱.....	298
反垃圾邮件.....	299
支持邮件的延展	307
本章所用模块.....	312
更多参考资料.....	313
第9章 目录服务	315
什么是目录?	315
Finger：一个简单目录系统.....	316
WHOIS目录服务	320
LDAP：一种复杂的目录服务	323
使用Perl进行LDAP编程.....	324

建立LDAP连接	324
进行LDAP搜索	327
条目在Perl里的表示	330
通过LDIF来新增条目	332
使用标准LDAP操作来新增条目	334
删除条目	335
修改条目名	336
修改条目属性	338
更深入的LDAP主题	340
综合练习	349
活动目录服务接口 (ADSI)	355
ADSI基础	356
从Perl调用ADSI	358
处理容器对象/集合对象	360
识别容器对象	361
那么如何了解关于对象的其他信息?	361
搜索	364
使用WinNT和LDAP名称空间执行常规管理任务	366
通过ADSI访问用户信息	367
通过ADSI访问组	369
通过ADSI处理文件共享	370
通过ADSI处理打印队列和打印任务	370
通过ADSI来处理基于Windows的操作系统服务	372
本章所用模块	373
更多参考资料	373
LDAP	373
ADSI	375

第10章 日志文件 376

读取文本日志	376
读取二进制日志文件	377
使用unpack()	377

调用操作系统（或其他）二进制文件.....	382
使用操作系统的日志记录API.....	383
日志文件数据结构.....	385
处理日志文件信息.....	387
日志信息的空间管理.....	387
日志解析及分析.....	395
创建自己的日志文件.....	425
日志记录的快捷方式和格式化帮助.....	426
初级/中级日志记录框架.....	427
高级日志记录框架.....	428
本章所用模块.....	430
更多参考资料.....	431
第11章 安全	433
注意不必要的或未授权的修改.....	434
本地文件系统的修改.....	434
网络数据的改变.....	440
关注可疑行为.....	442
本地的问题信号.....	442
发现问题模式.....	444
危险的网路，或者说“Perl挽救了局面”.....	448
防范危险行为.....	459
建议更好的密码.....	459
拒绝差密码.....	460
本章所用模块.....	465
更多参考资料.....	466
第12章 SNMP	467
从Perl中调用SNMP.....	467
发送和接收SNMP Trap、Notification和Inform.....	478
其他SNMP编程接口.....	481
本章所用模块.....	483

更多参考资料	483
第13章 网络映射和监控	485
网络映射	485
发现主机	486
发现网络服务	494
物理定位	496
展现信息	498
文本展现工具	498
图形展现工具	502
监控框架	516
对现有监控软件包进行扩展	518
现在还剩什么?	520
本章所用模块	520
更多参考资料	521
第14章 实验性学习	522
漫步时间线	523
任务一：解析crontab文件	523
任务二：显示时间线	524
任务三：输出正确的XML文件	526
思路汇总	527
总结：我们可以从中学到什么?	529
地理编码的乐趣	530
邮政地址的地理编码	530
IP地址的地理编码	533
总结：我们学到了什么?	536
与MP3打交道	536
总结：我们可以学到什么?	538
临别演出	539
第一步：用WWW::Mechanize从Wiki页面获取数据	539

第二步：解析数据	542
第三步：对数据做地理编码并画图	543
总结：我们可以学到什么？	546
记住娱乐	547
本章所用模块	547
本章中的资料来源.....	548
 附录A 8分钟XML教程	 549
附录B 10分钟XPath教程	554
附录C 10分钟LDAP教程	563
附录D 15分钟SQL指南	569
附录E 5分钟RCS教程	582
附录F 2分钟VBScript翻译到Perl教程	586
附录G 20分钟SNMP教程	592

前言

你需要那些能够帮你简化并高效完成系统管理工作的工具吗？来这里算是找对了。

Perl源于传统的系统管理工具箱而成为一种高效的编程语言。这些年来它不断适应并扩展，以跟上操作系统的发展，也能完成更多新任务。如果你懂一点Perl，并且想轻松完成系统管理任务，那么本书正是你需要的。相信各种程度的Perl开发人员和系统管理员都能在这本书中找到有价值的信息。

这个版本有什么更新？

为了提升这本书的价值，我们在第二版中对内容进行了大量改进。下面列出其中主要的更新：

新的书名

我和我的编辑都意识到，这本书的内容更注重的是如何使系统管理任务自动化，这样才能提高你的工作效率、带来更多乐趣。这其中虽然使用了Perl，但它只是工具，不是重点，真正的重点是自动化。

新的内容

说到这里真的有点不知从何说起。新版本添加了四个章节和两个附录，这些篇幅能占到原书的一半。这些新增的内容可以说是新工具和新技术的聚宝盆，想必你会喜欢。我在其中加入了所有（理想中的）系统管理的书应该有的内容，包括：XML和YAML的最佳实践（通过XML::LibXML、XML::Twig和XPath），处理配置文件，更加深入的LDAP主题（其中含有最新的Net::LDAP相关信息），邮件相关的主题（POP3/IMAP、MIME和反垃圾邮件），处理文件系统的新办法，更高级日志文件的创建和分析工具，DHCP，使用Nmap和其他工具来监控网络，网络数据包创建和嗅探，使用GraphViz/RRDtool/Timeline这样的图形化工具的信息汇报，使用SHA-2替代MD5，SNMPv3，Mac OS X，把VBScript代码转换成Perl，地理编码

(geocoding)，MP3文件处理，使用谷歌地图等等。

新的建议

这本书的部分价值在于它能向你传授那些有经验的系统管理员的做事方式。我尽可能把自己日常积累的小技巧（以及其他老手所分享的经验）都汇集在这本书中。所以新版本加入了更多的补充内容，用来解释为什么要介绍相关技术。

操作系统和软件信息的更新

所有的命令和代码都被更新以适应最新版本的基于Unix（包括Linux和Mac OS X）和Windows的操作系统。

模块和代码的更新/改进

本书对第一版介绍过的模块和相应的代码都进行了跟踪和更新。对于那些不能再使用的模块或者有更好选择的模块，本书提供了可以代替的模块。另外，所有的范例程序都可以通过“use strict”的限制。

勘误表

我尽可能更新了本书自第一本发行以来收到的所有勘误表。我非常感激那些读者给O'Reilly和我发来的错误报告，这使得我能在这一版修正那些问题。特别感谢Andreas Karrer，第一版的德语翻译。他仔细阅读了本书的每一个字句并且提交了大约200个问题（大多和排版相关），而且全都值得推敲。

本书的结构

本书的每一章都讨论了系统管理的一个领域，在结束时列出了那一章所使用的 Perl 模块和那些可以让你深入探索的相关资料。这些章节包括：

第1章 简介

本章进一步介绍本书涉及的主题，告诉你为什么要介绍它们，也告诉你如何从中获益。这本书的内容很强大，而且主要是针对那些权威用户（比如说Unix超级用户和基于Windows的操作系统管理员）的，所以这一章还介绍一些重要的概念来帮助你写出更加安全的Perl程序。

第2章 文件系统

本章论述保持多平台文件系统的整洁以及确保它们被恰当地使用。本章从介绍各种文件系统的主要差异开始；然后展示如何使用Perl进行文件系统的高效遍历；最后会看看如何用Perl管理磁盘限额。

第3章 用户账户

本章讨论用户账户如何在两种不同操作系统上显示自己，包括为每个用户账户存储哪些信息，以及如何用Perl来处理这些信息。这个主题后来延伸到使用Perl来开发

简易的账户管理系统。在构造这个系统的过程中，我们会查看在简易数据库中记录账户、创建账户及删除这些账户所必需的机制。

第4章 用户活动

本章讨论了如何对用户的活动进行自动化管理，包括跟踪并控制用户启动的进程、打开的文件和网络操作。这一章还介绍了一些与操作系统相关的管理框架和工具（比如Windows Management Instrumentation、图形用户界面安装工具、*Isot*等等），这些都有助于在各种平台上进行面向用户的任务。

第5章 TCP/IP名称和配置服务

网络命名和配置服务对于TCP/IP网络的主机来说是非常重要的通信基础设施。这一章从历史上曾经被使用的host文件开始介绍，经历了NIS（Network Information Service，网络信息服务），最终过渡到DNS（Domain Name Service，域名服务）这个Internet的黏合剂。每一步我们都展示了如何使用Perl来简化这些网络服务的专业管理。另外还介绍了如何用Perl来管理DHCP（Dynamic Host Configuration Protocol，动态主机配置协议）。

第6章 使用配置文件工作

几乎我们接触过的所有系统和软件包都需要配置文件才能正确工作。这一章介绍的工具能简化我们通过Perl来读写配置文件的任务。这里我们主要关注的是XML文件，另外还介绍了这方面的最佳实践。

第7章 SQL数据库管理

系统管理领域的一大趋势是越来越多地使用关系数据库系统。所以作为一个系统管理员，也有必要熟悉SQL数据库的管理。这一章介绍了Perl最引以为豪的SQL数据库框架——DBI，另外还列出了一些使用它来管理数据库的例子。

第8章 E-mail

这一章展示了Perl如何把电子邮件作为系统管理的工具。在讨论了通过SMTP发送（包括基于MZME的HTML消息）、通过POP3/IMAP接收、通过Perl来分析邮件内容之后，我们还介绍了几个有趣的应用程序，包括用来分析垃圾邮件和管理技术支持类邮件的工具。

第9章 目录服务

随着我们处理的信息量的增加，查询和维护的复杂度也逐渐增加，所以有必要通过目录服务来存取它们。系统管理员不仅需要使用这些服务，而且往往还需要管理这些服务。这一章介绍了一些常见的目录服务协议（或框架），比如LDAP和ADSI，另外也介绍了如何通过Perl来使用它们。

第10章 日志文件

系统管理员常常会淹没在日志文件的海洋中。每台主机、每个操作系统、每个程序

都可能有自己的日志信息。这一章分析基于Unix和Windows的操作系统上的日志系统，也介绍了如何更有效地分析日志信息。

第11章 安全

这一章关注的正是让人望而生畏的“安全”专题，通过演示你能看到如何用Perl来提高主机和网络的安全性。

第12章 SNMP

这一章关注的是SNMP（Simple Network Management Protocol，简单网络管理协议）。它展示了如何使用这个协议来与网络设备通信（主动轮询或被动通知两种模式）。

第13章 网络映射和监控

Perl提供了网络映射和网络监控方面的绝佳工具。这一章我们介绍了几种发现网络上的主机及其提供的相关服务的方法。然后我们还介绍了图形和文本模式的信息呈现方式，包括一些高质量的图形和图表绘制工具（比如GraphViz和RRDtool）。

第14章 实验性学习

这一章充满乐趣，所以阅读它的时候你大概不希望被老板发现。

附录

这本书的某些章节谈论的主题你可能并不熟悉。为了那些初学者，本书附带了一些迷你教程，以便读者快速入门。附录部分包括关于XML(eXtensible Markup Language,可扩展标记语言)、XPath (XML Path Language, XML路径语言)、LDAP (Lightweight Directory Access Protocol, 轻量级目录访问协议)、SQL (Structured Query Language, 结构化查询语言)、RCS (Revision Control System, 修订控制系统) 以及SNMP的介绍，另外还有把VBScript翻译成Perl的指南。

排版约定

本书中使用的排版格式如下：

斜体字 (*Italic*)

用于文件以及路径名称、用户名、目录名、程序名、主机名、URL以及首次出现的术语。

等宽字 (Constant width)

用于Perl模块名、函数名、名称空间、库、命令、方法、变量以及展示代码和终端输出时。

等宽黑体字 (**Constant width bold**)

用于表示用户输入的文字以及代码范例中着重显示的部分。

等宽斜体字 (*Constant width italic*)

用于表示命令行中需要用户自己替换为实际内容的部分以及代码注释。

关于操作系统名

这本书一贯关注多平台的系统管理。然而，如果总是必须说“微软Vista/微软Windows Server 2008/微软Windows Server 2003/微软XP脚本”，或者“Linux/Solaris/Irix/HPUX/Mac OS X等平台的脚本”，真的让人感觉乏味至极。在咨询了一些这方面的资深人士之后，我决定这样来统称这些操作系统：

- 在谈论微软的系列产品（微软Vista、微软Windows Server 2008、微软Windows Server 2003和微软XP）的时候，我把它们称为“基于Windows的操作系统”，起码在每一章的开头是这样称呼的。在章节的后面可能把它简化为“Windows”。如果涉及某些特殊的操作系统特性，我会使用它的全名。另外，本书的大多数代码都是在这些平台上测试过的。
- 在谈论Unix系列的操作系统的时候，我把它们称为“Unix”（这包括了Linux和Mac OS X）、“Unix系列”或者“Unix”衍生。如果涉及某种Unix厂商或者特殊发行版本，我会使用它的全名。

编码规范

下面是关于本书中代码的几点声明：

- 本来所有的范例程序的第一行都写有`use strict;`并且测试通过（我希望你也养成这样的习惯）。但是鉴于本书的例子数量众多，一再重复这一行显得太过刻板，所以我最终决定从例子中拿掉这些代码。不过，请确信这些例子都已通过了相关测试。
- 这里的绝大多数代码都使用了Steve Hancock卓越的`perltidy`实用工具（<http://search.cpan.org/dist/Perl-Tidy/>），从而增强了可读性。
- 这本书中的很多代码在我看了Damian Conway写的《Perl Best Practices》（O'Reilly出版）一书之后进行了重写，使代码质量尽可能达到这本书建议的程度。我强烈推荐你也读一下那本书，无论是从编码质量的角度，还是从提升基本开发水平的角度都应该考虑读一下。在我写本书时，受《Perl Best Practices》启发的自动化源码分析工具`Perl::Critic`还在开发中，所以我没能用上它。不过你应该能用上，它可

是一个非常卓越的工具。

范例代码的使用

本书的目的是帮助读者完成任务。总的来说，你可以在程序和文档中使用本书中的代码，对此不需要获得我们的许可，除非你复制了绝大部分的代码。比如，使用书中的几段代码来编写程序并不需要获得许可，但贩卖或者发布包含有O'Reilly书籍中例子的光盘则需要许可。引用本书和书中的范例代码来回答问题不需要许可，而将本书中大部分范例代码加入你自己产品的文档中则需要许可。

我们欢迎你在引用时提及来源，但并不强求。通常引用来源包括书名、作者、出版商以及ISBN。例如：“Automating System Administration with Perl, Second Edition, by David N. Blank-Edelman. Copyright 2009 O'Reilly Media, Inc., 978-0-596-00639-6”。

如果觉得自己对范例代码的使用超出了合理使用或者是上面许可的范围，请不吝与我们联系，邮件地址是`permissions@oreilly.com`。

如何联系我们

本书的内容都经过测试及验证，尽管我们做了最大的努力，但错误和疏忽仍然是在所难免的。如果你发现在什么错误或对将来的版本有什么建议，请通过如下地址告知我们：

美国：

O'Reilly Media, Inc.
1005 Gravenstein Highway North
Sebastopol, CA 95472

中国：

北京市西城区西直门南大街2号成铭大厦C座807室（100035）
奥莱理技术咨询（北京）有限公司

O'Reilly的每一本书都有专属网站，你可以在那找到关于本书的相关信息，包括勘误表、范例代码以及其他的信息。本书的网站地址是：

<http://www.oreilly.com/catalog/9780596006396/>

询问技术性的问题或对本书进行评论，请发送电子邮件到：

`bookquestions@oreilly.com`

关于本书的更多信息、会议、资料中心和O'Reilly网络，请访问以下网站：

<http://www.oreilly.com/>

<http://www.oreilly.com.cn/>

第一版出版致谢

为避免前言变得类似奥斯卡获奖人的发言，下面只列出压缩过的第一版出版致谢。

感谢Perl社区，尤其是Larry Wall、Tom Christiansen和如云般的程序员和黑客们，感谢你们倾注了大量的时间和精力到这个语言，也感谢你们热心地分享程序、经验和热情给Perl社区的每一个人。

感谢系统管理员社区，包括Usenix、SAGE以及这些年来参与LISA会议的那些人。感谢Rémy Evard带我进入这个领域，又和我成为朋友，你是我个人的“行为榜样”。如果我还能不断提高，将来一定会成为你那样的系统管理员。

感谢本书第一版的审阅者：Jerry Carter、Toby Everett、Aleen Frisch、Joe Johnston、Tom Limoncelli、John A. Montgomery, Jr., Chris Nandor、Michael Pepler、Michael Stok以及Nathan Torkington。

感谢O'Reilly的朋友们：感谢Rhon Porter的插图，感谢Hanna Dyer和Lorrie LeJeune设计的惊艳封面动物，也感谢O'Reilly的产品部门。我还得感谢Linda Mui，第一版的编辑，有了你的熟练技巧和关心我才能顺利“诞生下”这本书，还能让他逐渐成长。

感谢我的心灵同伴：Somerville的Havurat Shalom。感谢你M'kor HaChayim，为这本书，也为你在我生命中带来的所有祝福而感谢你。

感谢津巴布韦的绍纳族那卓越的安比拉琴音乐。

感谢我的朋友们（Avner、Ellen、Phil Shapiro、Alex Skovronek、Jon Orwant和Joel Segel），感谢美国东北大学计算机和信息科学学院的所有教职员。也要感谢我的老板Larry Finkelstein，你就是学院的明星。

还要感谢我的小家庭（Myra、Jason和Steven Edelman-Blank），我的猫Shimmer和Bendir（再见Bendir，我很想念你），还有我的TCM后勤（Kristen Porter和Thom Donovan）。

谨以本书第一版献给Cindy，我一生的挚爱。

第二版出版致谢

如果说和那些美好的人相识是好的，那么能一直和他们同在则是更好的。我会一直感谢和本书第一版有关的所有人，另外还要加上下面这些更新：

这一版相较于上一版有豪华的技术审阅团队。我非常感谢Aleen Frisch、Aaron Crane、Aleksey Tsalolikhin、Andrew Langmead、Bill Cole、Cat Okita、Chaos Golubitsky、Charles Richmond、Chris Grau、Clifton Royston、Dan Wilson、Dean Wilson、Denny Allain、Derek J. Balling、Earl Gay、Eric Sorenson、Eric Toczec、Federico Lucifredi、Gordon “Fyodor” Lyon、Graham Barr、Grant McLean、Hugh Brown、James Keating、Jan Dubois、Jennifer Davis、Jerry Carter、Jesse Vincent、Joe Morri、John Levine、John Tsangaris、Josh Roberts、Justin Mason、Mark Bergman、Michel Rodriguez、Mike DeGraw-Bertsch、Mike Stok、Neil Neely、Petr Pajas、Philip J. Hollenback、Randy Dees、Scott Murphy、Shlomi Fish、Stephen Potter、Steve Atkins、Steven Tylock、Terry Zink、Thomas Leyer、Tim Bunce、Tobias Oetiker、Toby Ovod-Everett和Tom Regner在阅读和改进本书上花的精力。我总是会为系统管理和Perl社区的慷慨及热心感到自豪。

这本书的编辑众多，所以只能一并感谢你们。自本书的第一版以来，先后参与的编辑有：Linda Mui、Paula Ferguson、Nathan Torkington、Allison Randal、Colleen Gorman、Tatiana Apandi、Isabel Kunkle和Andy Oram。我还得感谢那些幕后推动本书的O'Reilly朋友们：Mike Hendrickson、Rachel Head、Sarah Schneider、Rob Romano、Sanders Kleinfeld和所有人。

起初我对海獭还是比较陌生的，直到第一版发行之后我变得越来越喜欢它们。它们有那么多值得爱护的地方，不过人类却往往很难善待它们。我们的生活方式威胁到它们的生存，这就是为什么它们仍然被列为濒危动物。我相信它们值得我们保护和支撑。我已经加入了一个名叫Friends of the Sea Otter的组织，它的总部在加利福尼亚的蒙特利。我也鼓励你加入这样的组织。

安比拉琴音乐让我在写作本书上一版的时候保持神智清醒状态。在写这一版的时候，我需要通过瑜伽来保持健康。我在这里要对我的老师Karin Stephan以及她的老师B.K.S. Iyengar致谢，感谢她们与我分享这种让身体和意念合一的方法。

我已经努力缩减第一版的感谢致辞，所以希望你能再忍受我一会儿。对我来说，第一版发行之后，生命中最大的改变就是大儿子Elijah的诞生。他对我们来说是永远的祝福，且不论这里的“祝福”是动词还是名词。

在我的家乡，有些公交线路走的是无轨电车。有一天，我向一个电车司机打听一条不熟悉的路，没想到他告诉我：“对不起，我不知道这条路在哪里，我只是按照电线的走向开车而已。”

我想这句话绝不会出自于一个好的系统管理员，因为系统管理是一门艺术，而不是墨守成规。系统管理（或网络管理）这个工作好比设计线路的类型和走向，部署并监控线路，可能有时还得拆除线路并重新设计。系统管理几乎没有什么可以循规蹈矩的，在需求多变的跨平台环境更是如此。类似于其他的那些手艺，水平高低的区别就在于解决问题的简洁程度。不论你是一个全职的系统管理员还是偶尔为之的玩家，这本书都会让你在这条路上走得更加轻省。

自动化是必须的

几乎可以这么说，任何手动管理系统的解决方案都是错误的。所以这本书就是致力于让你摆脱手动维护的烦恼。

即使在最好的经济形势下，系统管理都不是一个清闲的工作。对于那些主动选择这份工作的人或被老板调入这个职位的人来说都是如此。常常有这样的情况，老板不批准录用新人，于是那些没有经过培训的老员工就被赋予了管理系统的职责。

在这种情况下，有效的自动化正是一种灵丹妙药。它能有效提高工作效率，把用于系统管理的单调劳动时间节约出来，这些时间可以用来做更多有趣的事情，甚至可以陶冶自己的情操。

我和编辑协商之后修改了这一部分的标题，因为我们意识到这部分内容的重点在于通过自动化来提高工作效率。在这本书里，我会努力推荐那些有用的工具（包括改变思维方式的工具）来让工作更加有效。在最后一章你会看到，甚至连娱乐的效率也能提高。

相关话题：配置管理

在开始之前有必要提醒大家这本书的目的。这不是一本关于配置管理的书，也没有涵盖配置管理中常用的工具，比如`cfengine`、`puppet`和`bcfg2`。

绝大多数环境可以从机器和网络的配置管理中获益，也可以从日常流程自动化中获益。这本书的重点在第二个方面，但是我也鼓励你去看看上一段中提到的那些配置管理工具。一旦熟悉了那些工具，就可以配合这本书将要介绍的脚本来使用。

Perl如何帮助你

系统管理员应该能够使用任何计算机语言来完成任务，那么为什么用整本书来讨论Perl？

这个问题的解答需要回到系统管理的本质。一个名叫Rémy Evard的同事和朋友曾经这样描述系统管理员这个职务：

一方面，你有一系列的资源：计算机、网络、软件等等；另一方面，你也有一些有需求和项目的用户——想完成工作的人。我们的任务就是让资源和用户完美地融合，努力沟通需求和技术两个世界。

系统管理常常是一种集成工作，而Perl恰好是最著名的胶水语言。早在万维网时代之前Perl就已经广泛地被系统管理员使用了，而这些年来万维网的普及也带动了对胶水语言的需求。最近几年在LISA（Large Installation System Administration，大规模安装系统管理大会）和另外的一些聚会上得到的反馈就是Perl仍然是在这个领域的主要语言。

Perl的有些方面对系统管理来说是非常有用的：

- 它从一开始就符合很多Unix shell和C语言的习惯，而这两者对于大多数系统管理员来说都是非常熟悉的。
- 它在大多数比较新的操作系统中都是可用的，并且它还尽可能地提供了一致的接口。这对跨平台的系统管理来说非常重要。
- 它有绝佳的文本处理功能，也擅长处理数据库和网络编程问题，这三个领域恰恰是这个职业的重要根基。
- 语言的核心部分可以较为简单地通过模块化方式进行扩充。
- 广大专业的用户社区投入了无法计算的时间来为绝大多数任务创建了模块。这些模块大多被有条理地收集起来（之后有详细的介绍）。社区的支持常常会带来意外的

惊喜。

- 即使对纯粹的编程来说，它也足够有趣。

在罗列了众多好处之后，也必须指出Perl的一些局限性。它并非能解决每个问题，有时甚至不是系统管理编程中最好的工具。在有些人看来，它还有些不足：

- Perl的面向对象编程机制常常让人捉摸不透。在这方面Python和Ruby有更好的名声。
- Perl并非总是那么简单一致，往往充满了奇怪的调用。而在其他语言中则没有这么多让人惊讶的地方。
- Perl非常强大，但也容易走火。

这里还是建议你选择合适的工具。当然，对我来说，Perl总是合适的工具，所以它成为了这本书的重点。

这本书会带你学会

在1996年到1998年的《蝙蝠侠》电视剧中，英雄拍档的腰上都带着工具。在蝙蝠侠和罗宾要爬入某个建筑时，蝙蝠侠会说“罗宾，快用蝙蝠抓钩！”或者“罗宾，快用蝙蝠眩晕气体！”，反正他们手边总是有合手的工具，可以用来征服坏蛋。这本书正是帮你佩戴好那些装备，用来应付系统管理中的那些难题。

每一章都提供了三个内容：

简洁的系统管理专题信息

每一章都深入介绍一个系统管理专题。用一本书来容纳所有跨平台系统管理的主题是不可能的，因为那样篇幅就没法控制。Unix系统管理领域广受好评的两本书——Aleen Frisch的《Essential System Administration》（O'Reilly出版），Evi Nemeth、Garth Snyder、Scott Seebass和Trent H. Hein合著的《Unix System Administration Handbook》（Prentice Hall出版）的篇幅分别是本书的两倍和三倍，而且我们关注的是三种迥异的操作系统：Unix（包括Linux这样的衍生）、基于Windows的操作系统和Mac OS X。

尽管专题的选择不可能让大家都满意，但我们还是精心挑选，努力把系统管理（和网络管理）中最有价值的信息呈现给各个层次的读者。老手和新人可能会学到不同的内容，但应该有每个人值得仔细揣摩的地方。每章的结尾都有参考资料，有助于读者更加深入地了解相关主题。

书中有些主题对某些读者来说可能比较陌生，或者是缺乏一些背景经验，为此我编

写了附录来帮助大家快速入门。对于老手来说，这些附录往往也能扩展知识面，比如了解某个功能在其他平台上是如何实现的。

系统管理中常用的Perl技巧和方法

想要从这本书中学到东西，你得有些基本的Perl经验。本书每一章都会有或简单或复杂的Perl代码，当然我们会对那些相对复杂的技巧、数据结构或者俗语作深入介绍。我相信在学习的过程中，不论是哪个层次的Perl编程人员都能逐渐积累Perl技能和经验并在以后的编程中使用。而且随着水平的提高，你应该可以反复从书中发现新的Perl乐趣。

为了提高学习的乐趣，我会常常用多种方法解决同一个问题，而不是抛出单一的解答。请记住Perl的座右铭：“条条大道通罗马”。这些解答是为了给你的Perl腰带挂上更多的工具，而更多的工具往往意味着面对新问题时有更好的选择。

虽然有时候一种技术显然胜过另外一种，但我们还是尽可能给出更多选择。请记住，这本书只能带你入门，因为你可能会面对更加复杂的问题。在某个问题中显得生涩的技术，可能在另一个问题中会大有用处。另外请多包涵，我会就每种问题对比各种技术的优劣，有时还附带介绍我个人的喜好。

系统管理的最佳实践和原则

这章开头的时候我说过，系统管理是门艺术，会有好坏的差别。这25年来，我在一些苛刻的多平台环境中工作，并且担任了系统管理和网络管理的职务。所以在每一章里，我都会把这些年的经验融入到范例中，总结出最佳实践和原则。我有时会用个人的实战经历作为引言，带你逐渐深入了解整个主题。希望你在阅读时能学到系统管理这门艺术的精髓，最终进入豁然开朗的阶段。

你需要什么

要想从这本书中学到东西，你得有一些技术背景和资源。我们先讨论背景：

你得懂点Perl

这本书没有花篇幅在Perl语言基础上，所以你需要找些其他的资料来补充这部分内容。相信Randal L. Schwartz等人合著的《Learning Perl》（O'Reilly出版）的这类书能带你轻松上路。

你得了解操作系统的基础

这本书假定你对需要管理的操作系统已经有基本的了解。你起码得知道如何在操作系统里面工作（运行命令、查找文档等等）。那些操作系统内置的复杂框架的背景知识（比如Windows的WMI或者SNMP）我们会在书中介绍。

你得了解操作系统的特性

我会努力介绍主要的操作系统之间的差异，但是不可能覆盖所有的差异。其实每个Unix的衍生系统都有自己的特色。所以你应该深入了解相关的操作系统特性，并且能够对细微的差异融会贯通。

对于技术资源而言，你需要两样东西：

Perl

你得确保需要管理的每个系统都安装了Perl。如果没有安装，你可以到Perl网站的下载页面，找出适合特定操作系统的源代码或者二进制版本的Perl。这本书中的代码在标准Perl 5.8.8版本和ActivePerl (5.8.8) 822版本下开发和测试。后面还有更多关于版本的介绍。

查找和安装Perl模块的能力

后面的某个章节专门讨论了如何查找并安装Perl模块，这对我们来说是至关重要的技能。这本书假定你已经有了安装模块的技能和系统权限。

每一章的末尾都列出了本章代码中所涉及到的模块版本。其中的版本信息是必要的，因为并非每个模块都能保证向后兼容。如果运行时遇到问题，版本信息有助于你判断书中的模块是否更新过。

关于本书使用的Perl版本的解释

我选择将本书中的代码都在标准Perl 5.8.8和ActivePerl (5.8.8) 822下进行开发和测试。这个选择可能带来以下的疑问。

为什么不用Perl 5.10?

Perl 5开发团队已经成功完成了Perl 5.10版本的开发。他们在这个版本中对语言进行了卓越的改进，我建议你去尝试一下。但是Perl 5.10并不是本书出版时最成熟的版本，实际上写作本书的时候还没有任何主流操作系统带有这个发行版本。因为我知道每个新版本的推广都有很长的时间，所以我不希望书中的代码依赖于那些绝大多数人无法立即可用的语言中的特性。书中的代码在Perl 5.10版本中应该都能正常执行，选择稳定版本出于我想照顾大多数读者的需要。

为什么没介绍Strawberry Perl?

Strawberry Perl (<http://strawberryperl.com>) 是Win32平台上特殊的Perl发行版，它更加接近标准Perl，也更加“自给自足”。ActiveState公司的Perl发行时带有PPM这个包管理

系统，所以用户不必通过CPAN(Comprehensive Perl Archive Network,Perl综合典藏网)来编译或升级模块。Strawberry Perl则致力于提供一个让编译和CPAN的使用简单（或至少有可能）和规范的环境。

我认为这是一个卓越的项目，因为它真的可以帮助非Win32系统的Perl社区了解如何提高可移植性。目前来说这个项目已经有了可喜的进展，不过在写作本书的时候它还显得比较稚嫩（比如缺少很多Win32::模块）。这也让我最终决定了放弃此版本，不过它真的值得关注。

那么Perl 6呢？

哦，这真是个重要的问题，不是么？我很有幸与Jesse Vincent（Perl 6的项目经理以及鼎鼎大名的RT问题跟踪系统的作者）聊起过这个话题，以下是他所说的Perl 6的近况：

Perl 5是一个成熟而且被广泛接受的产品化语言。Perl 6正在快速成熟，但是目前还没有达到产品级别的成熟度。

有些Perl 5模块已经可以让你提前尝试Perl 6计划实现的特性（当然有些已经进入了Perl 5.10）。我鼓励你去尝试类似Perl6::Slurp或者Perl6::Form这样的模块。但是到目前为止，Perl 6并没有产品级别的实现，因此本书不会提及。然而，当Perl 6足够成熟的那一天，它带来的那些新兴模块会对本书提到的模块带来冲击。所以我想到了那个时候，我也许会把这本书改写成Perl 6版本的。

如何在Windows Vista中使用范例代码

本书中的代码都在微软Windows Vista中测试过，但是要在在这个平台上正确执行代码需要些小技巧：请注意要使用提升权限（elevated privilege）来执行。但是很难说哪些代码要用这个技巧，哪些不用。例如第2章中Windows限额的范例，有些能直接运行，而有些如果没有以提升权限执行则会失败，而且错误信息几乎无法读懂。

在Vista的UAC(User Account Control，用户账户管理)管理下，仅以管理员（Administrator）身份来运行代码并不够，你还必须显式地以提升权限级别运行它。下面列出我所知道的Perl脚本运行方法（主要是因为不能右键调出“Run as administrator”）。你可以选择在自己的环境下最有效的方法：

- 使用`runas.exe` 命令行工具程序。
- 设置`perl.exe`程序以管理员身份运行（右击可执行程序，选择“Properties”，然后切换到“Compatibility”标签页并选择“Run this program as administrator”）。

- 使用Elevation Power Toys(参考<http://technet.microsoft.com/en-us/magazine/2008.06.elevation.aspx>和<http://technet.microsoft.com/en-us/magazine/2007.06.utilityspotlight.aspx>)来将Perl脚本执行的身份设置为系统管理员。
- 使用`pl2bat`命令行工具程序把Perl脚本转化成批处理程序，然后授予批处理程序以管理员身份执行的权限。批处理程序并不需要类似前面所用的那些特别的方法就能奏效。

你可能在想是否有特殊的方法能从Perl脚本中申请这个特殊的提升权限，但是Jan Dubois (Windows Perl名人) 已经给出了否定的答案。他的说法是，已经运行的程序无法自行升级权限，只能在启动的时候被授权。最接近的解决方案是，使用Win32模块的`IsAdminUser()`函数来检查程序启动时是否已经获得了这样的权限。如果没有就使用其他东西（如`runas.exe`）来重新启动这个脚本。

最后介绍一脉相承的技巧：在很多章节里面我推荐使用微软的Scriptomatic工具来熟悉WMI。默认情况下这个工具也不能在Vista中运行，因为它也需要提升权限才能工作，而它是个“HTML应用”（`.hta`）文件。类似于Perl脚本，`.hta`文件要获得管理员权限并非易事。

下面就是绕过这个限制，让你能够使用这个卓越工具的技巧：

1. 右击任务栏上的IE图标，选择“Run as administrator”，这样就能使得浏览器以提升权限运行。（注意：千万别用这个特殊身份运行的IE来浏览网页，也不要用它打开Scriptomatic文件之外的其他文件，这样才能确保系统安全。）
2. 按下Alt键来显示IE的文件菜单，选择“Open...”，然后按下“Browse...”按钮，选择文件类型为“All”，然后打开Scriptomatic 的`.hta`文件所在的目录，最后选中那个文件，这样就大功告成了。

载入和使用模块

使用Perl来管理系统的最大好处在于可以使利用模块化的免费代码。这本书中提到的模块可以在以下三个位置找到：

Perl综合典藏网 (CPAN)

CPAN是一个巨大的Perl源代码仓库，也是文档、脚本和模块的仓库，在全球数百个站点都有镜像。关于CPAN的信息可以访问<http://www.cpan.org>。最简单的搜索CPAN模块的方式是访问<http://search.cpan.org>。使用“CPAN Search”搜索框常常能找到适合的模块。

预编译包的独立仓库

马上我们会用到PPM(Perl Package Manager, Perl包管理器), 一个对Win32 Perl用户来说至关重要的工具。这个工具能从仓库(最著名的应该是ActiveState所运营的那个)获取预先编译好的模块包。关于后台仓库的详细列表可以访问<http://win32.perl.org>。如果我们用到的Win32包不能从ActiveState下载, 我会特别列出它的来源。

独立站点

有些模块没有发布到CPAN或其他任何的PPM仓库。我会尽可能地避免使用它们, 但是万一没有办法找到更好的替代模块, 我会告诉你怎么获得它们。

一旦获取之后, 如何安装这些模块呢? 这个问题的答案和目前使用的操作系统相关。现在的Perl发行版本都带有一个名叫`perlmodinstall.pod`的文件用来描述安装模块的事项, 你可以用`perldoc perlmodinstall`命令来阅读此文件。下面我会介绍如何为书中涉及的每种操作系统安装模块。

在Unix上安装模块

绝大多数情况下, 步骤如下:

1. 下载模块并解包。
2. 运行`perl Makefile.PL`命令来创建必需的`Makefile`。
3. 运行`make`来编译模块。
4. 运行`make test`来运行作者所写模块中包含的测试套件。
5. 运行`make install`来把模块安装到系统常用的位置。

如果你懒于手动键入这些命令, 请使用Andreas J. König所写的CPAN模块, 或者Jos Boumans所写的CPANPLUS模块。对于CPAN来说, 以下的命令就能奏效:

```
% cpan
cpan[1]> install      modulename
```

而CPANPLUS需要你这样做:

```
% cpanp
CPAN Terminal> i      modulename
```

两个模块都能聪明地解决模块依赖性问题(也就是说, 如果一个模块要求另一个模块运行, 那么将自动为你安装这两个模块)。它们都有内置的搜索命令用于搜索相关的模块。我建议你在系统中键入`perldoc CPAN`或者`perldoc CPANPLUS`来了解这两个模块中隐

藏的有趣功能。

在Win32上安装模块

在Win32平台上使用ActiveState的用于Unix的发行版镜像安装模块的过程有一个额外步骤：运行PPM。如果你有足够的耐心手动安装模块，可以使用类似于WinZip这样的程序来解压缩，然后使用`nmake`而不是`make`来编译并安装模块。

有些模块需要调用C语言编译器来完成安装，而相当多的Win32世界中的Perl用户都没有在电脑上安装必要的编译器，所以ActiveState创建了PPM来发布预编译的模块。

PPM非常类似于CPAN模块。它用一个叫做`ppm.pl`的Perl程序来完成从PPM仓库下载并安装特定归档文件的工作。你可以通过键入`ppm`或者通过运行来自Perl的`bin`目录中的`ppm-shell`来启动程序：

```
C:\Perl\bin> ppm-shell
ppm 4.03
ppm> install      module-name
```

PPM和CPAN一样，能用来搜索并安装模块。在`ppm>`命令提示符下输入`help`可获得简单的使用说明。

要成为万能的并不容易

在继续阅读之前，我们先用几分钟的时间来说些警戒之言。系统管理所用的程序往往会不同于其他的程序：在Unix或Windows上它们常常需要通过提升权限（`elevated privilege`）来运行，如以`root`或者Administrator身份运行。这样的权限同时也带来了责任，也就是说我们编程的时候要特别注意代码安全，因为一点点小的疏忽都可能导致重大损失。还有可能因为我们的不谨慎，导致代码中隐藏的漏洞被那些不太“圣洁”的用户恶意利用。下面列出的就是那些需要特别注意的情况。

尽量避免提升权限

无论如何，使用Perl是对的，但是要尽可能地避免让代码在提升权限状态下运行。毕竟绝大多数的任务并不需要`root`或者Administrator权限。

比如说，日志分析程序可能并不需要以`root`运行，建议你创建一个低权限的用户来支持这个任务。应该从高权限用户的专用程序中产生数据，然后再用普通权限的用户程序来分析这些数据。

尽可能早地放弃特权

有时候很难避免以`root`或者Administrator身份运行脚本，例如邮件发送程序就可以以系统中任何用户的身份来写文件。这种类型的程序应该尽可能早地退出全能模式。

Perl程序在Unix环境下可以设置`$<`和`$>`变量：

```
#永久放弃权限
($<,$>) = (getpwnam('nobody'),getpwnam('nobody'));
```

这样可以把实际和有效的用户ID设置为`nobody`，这是一个在绝大多数Unix/Linux系统上都存在的低权限用户（如果没有这个用户也可以自己动手创建）。要做得更彻底，还可以通过`$()`和`$()`来设置实际和有效的组ID。

Windows并没有用户ID的概念，但是也有类似的降低权限的方式，而且你还可以用`runas.exe`来切换执行用户的身份。

小心读取数据

在读取重要的数据（比如配置文件）的时候，先测试文件是否安全。比如，你可能要确定文件和文件所在的各级目录都是不能随意写入的，否则可能读到的是某人篡改过的版本。在Tom Christiansen和Nathan Torkington 合著的《Perl Cookbook》(O'Reilly出版)这本书中有个很棒的技巧可以实现这个测试。

另外的顾虑还有用户的输入。千万别假定用户输入的数据都是善意的。哪怕你在屏幕上输出Please answer Y or N:这样的文字，用户也可能会输入2 049个任意字符。没人知道这是出于恶意，还是因为用户临时离开终端而他两岁的小宝宝正好在敲打键盘。

用户输入可能会是很多奇怪问题的根源所在。我喜欢的例子是“poison NULL byte”这个著名的Perl CGI漏洞。（这一章末尾的参考文献中会列出出处，请务必仔细阅读！）这个漏洞利用了Perl在处理字符串中的NULL（\000）字节上和C库的差异。对于Perl来说，这个字符和其他字符并没有什么差别，但是对库程序来说，这意味着字符串的结束。

说得更加明白些，这意味着用户可以很容易地逃过安全检查。例如文章里面提到的密码修改程序，其中的代码可能如下：

```
if ($user ne "root"){ <call the necessary C library routine> }
```

如果恶意用户能把`$user`设置成`root\000`（也就是`root`后面跟着NULL字节），测试程序会认为用户名不是`root`，从而允许Perl脚本继续执行下去。但是当字符串被发送到底层的C库的时候，却会被当成`root`来执行，这样用户就绕过了安全检查。而且同样的手段

还可以用来访问系统上的任意的文件或资源。最简单的避免类似安全问题的手段就是用如下的代码来清洁用户的输入：

```
$input =~ tr/\000//d;
```

更好的办法是从用户输入中解析出正确的数据（例如使用正则表达式）。

注意：这只是用户输入导致问题的一个例子。因为用户输入可能导致如此多的问题，所以Perl有一套预防机制，称为*taint mode*（污染模式）。请参考Perl自带的perlsec手册中关于“taintedness”的精彩介绍。

小心写数据

如果你的程序可以写入文件或追加到文件末尾，请特别注意如何写入、写入到哪里以及何时写入。对于Unix来说，这非常重要，因为符号链接使得文件的交换和重定向变得非常容易。如果编程时不注意，可能会意外地把数据写到错误的文件或设备中。有两类程序需要特别注意这个问题。

第一类是往其他文件末尾添加数据程序。在尝试添加进文件之前请检查：

1. 在打开文件之前先查看文件的属性，使用stat()和常用的文件测试操作符来判断。避免对软链接或者硬链接进行操作，另外也要查看文件的读写权限和拥有者等。
2. 打开文件并添加内容。
3. 对打开的文件句柄调用stat()函数。
4. 比较第1步和第3步的数据，确保打开的文件句柄指向的是正确的文件。

第10章的bigbuffy程序展示了这个流程。

第二类是使用临时文件或者目录的程序。例如下面这段常见的代码：

```
open(TEMPFILE,">/tmp/temp.$$") or die "unable to write /tmp/temp.$$!\n";
```

这在多用户的机器上并不十分安全。因为\$\$变量所代表的进程编号在大多数机器上易于预测，这意味着你的临时文件名也易于预测。如果其他人提前创建了这个文件，那往往会是一个悲剧的开始。

最简单的措施就是使用Tim Jenness的File::Temp模块。这个模块从Perl 5.6开始就有了，使用方法如下：

```
use File::Temp qw(tempfile);
```

```
# 返回打开的文件句柄和相应的文件名
my ($fh, $filename) = tempfile();
print $fh "Writing to the temp file now...\n";
```

File::Temp模块还能帮你自动删除临时文件，有兴趣的话请查看模块的说明文档了解更多细节。

避免竞争条件

如果可能，请避免写出那些可能会落入竞争条件试探（race condition exploit）的代码。典型的竞争条件问题往往会假定如下的顺序是无懈可击的：

1. 你的程序先积聚一些数据。
2. 然后程序开始操作那些数据。

例如下面的例子：

1. 你的程序检查了一个bug报告文件的时间戳，以确定自从上次读入文件后没有添加任何内容。
2. 然后程序修改文件内容。

如果用户设法在中间加入额外的环节（可以称为“第1.5步”），并且做了一些重要的内容替换，那么就可能导致问题。如果用户可以让你的程序在第2步操作的数据和第1步的不同，那么他们就成功地完成了竞争条件试探（或者说抢在你的前面控制了数据）。其他的竞争条件可能会在处理文件锁定失误时发生。

竞争条件常常发生在扫描文件系统的系统管理程序中，这些程序先遍历检查所有的文件，然后再遍历修改文件。恶意用户往往能在两次遍历之间替换文件，这样会使修改发生在错误的文件上。请注意避免这种类型的问题。

保持乐观

必须提醒的是，系统管理是一件有趣的事情，并非大多数人想象的那样总是要和困难的问题打交道，其实也有值得享受的乐趣。搭建平台和支持别人是很有成就感的事情，你编写的Perl程序能让许多人受益，这就是非常美好的景象。

是时候回到主题了。

更多参考资料

<http://www.dwheeler.com/secure-programs/>是由David A. Wheeler 编写的供 Linux 及 Unix

安全编程人员阅读的HOWTO文档。其中介绍的概念和技术也同样适用于其他编程领域。

<http://nob.cs.ucdavis.edu/bishop/secprog/>包含了来自安全专家Matt Bishop所提供的关于安全编程方面的丰富资源。

<http://www.homeport.org/~adam/review.html>列出了由Adam Shostack提供的安全代码审阅纲要。

<http://www.canonical.org/~kragen/security-holes.html>虽然有些老旧，但仍然不失为一份优秀的指导如何探寻安全漏洞（特别是自己编写的代码内的安全漏洞）的文章，由Kragen Sitaker编写。

《Perl CGI Problems》一文由rain.forest.puppy撰写（《Phrack Magazine》，1999），阐述了CGI方面的安全问题和漏洞。可以在线阅读<http://www.insecure.org/news/P55-07.txt>，或者到Phrack的官方网站阅读<http://www.phrack.com/issues.html?issue=55>。

《Perl Cookbook》（Second Edition），由Tom Christiansen和Nathan Torkington合著（O'Reilly出版），有许多不错的关于安全编程的小秘诀。

第2章

文件系统

Perl用于拯救数据

慢镜头切入：我在写本书第一版的时候，笔记本电脑从桌上滑下来，重重摔在地上。当我捡起来的时候，发现它还没有摔烂，于是我打开电脑检查有什么异常。这时候它运行得非常慢，磁盘开始不规则地发出嘎嗒声。我把电脑关闭，希望这样能解决速度问题，因为软件错误也可能导致这样的问题。但屏幕怎么也不黑，系统不能正常关闭，这可不是个好兆头。

更加糟糕的是，电脑终于重新启动了，然而每次Windows NT启动时都会显示错误“file not found”。我意识到摔坏了磁盘。也就是说，磁头可能重重地碰到了磁盘柱面，而且恰好损坏了启动系统时需要用到的文件和目录。现在问题是：我的工作文件还完好么？进一步说，这本书的草稿文件还能恢复么？

我首先尝试启动电脑上的另外一个操作系统Linux，启动成功了，这是一个好迹象。然而我需要的文件是在Windows NT的NTFS分区。于是我使用了Martin von Löwis的Linux NTFS驱动（参考<http://www.linux-ntfs.org>，现在的Linux内核已经自带了），分区成功挂载上来，好像我所有的文件都在。

我试着把文件从磁盘分区中拷贝出来，一开始这个尝试还是比较顺利的，但是很快就在某个文件上卡住了：磁盘又开始发出嘎达嘎达的声音，拷贝只能以失败告终。其实，如果我能设法避免拷贝那些已经损坏的文件就能继续拯救其他数据。我用来拷贝数据的程序(*gnutar*)其实能够跳过某些文件，问题是我该如何知道哪些文件已经损坏。要从系统中16 000^[注1]个文件中找出那些被损坏的文件并不是一件容易的事情，一遍遍地运行*gnutar*显然不是办法，于是我决定考虑使用Perl。

注1： 当年的16 000个文件看来已经算很多了，然而我现在写这本书用的笔记本电脑中却有1 096 010个文件。所以我觉得如果这个故事发生在今天会更加有趣。

这一章稍后的部分我会向你展示这段代码。为了让代码易于理解，我先大致介绍一下文件系统的背景和用来访问文件系统的Perl代码。

文件系统差异

我们先快速介绍几种与操作系统相关的文件系统。有些知识对你来说可能已经不那么新鲜了，因为你已经对其中一种操作系统有了深入的了解。然而深入了解文件系统之间的差异还是有好处的，对于那些要在多种平台上进行开发的Perl程序员来说更是如此。

Unix

所有现代的Unix衍生版本都带有某种类似于Berkeley快速文件系统（Fast File System, FFS）的文件系统，因为这是它们共同的祖先。每个发行商都用不同的方法来拓展它：有的系统在安全性方面加入了POSIX访问控制列表（Access Control List, ACL）支持，有些则为了提高可恢复性而加入了日志支持，还有一些加入了特殊文件属性的支持。我们主要针对那些最常见的基础特性来编码，这样的功能在各种Unix平台上都可以使用。

Unix文件系统的顶端（或者说`root`）是用单个斜线（`/`）来表示。想要唯一地表示文件或目录，我们一般都会在路径开头加上斜线，然后跟上一串目录，越后面的目录会在越深的层次中。路径的最后才是人们真正感兴趣的文件名或者目录。目录和文件的名字一般都是区分大小写的，几乎所有的ASCII字符都可以用来命名文件，但往往处理起来需要某些技巧，所以建议你还是把文件名限制在字母、数字和少数几个标点符号内。

基于Windows的操作系统

如今所有基于Windows的操作系统都支持三种文件系统：FAT（File Allocation Table，文件分配表），NTFS（NT File System，NT文件系统）以及FAT32（FAT的改进版，支持大分区和更小的簇）。

Windows的FAT文件系统与DOS里面的版本相比有些改进。不过我们可以先看看最典型的FAT系统的一些特性。不论是在基础模式（basic mode）还是在实模式中（real mode），FAT文件系统都遵从8.3的文件名规范，这意味着文件名和目录名最多只能有8个字符的长度，然后必须是一个点号（也就是英文句号），最后是最多3个字符长的后缀。在Unix中，句号并没有特殊的含义；然而在FAT文件系统中，文件名只能也必须带有一个句号，用来分隔文件名和后缀名。

实模式FAT系统后来有一个名叫VFAT的扩展版本，也可以称为保护模式（protected-mode）FAT。这也是绝大多数情况下实际被使用的FAT版本。VFAT隐藏了文件名限制，

它其实是通过一个有创意的hack来支持长文件名：VFAT使用一组标准的文件名（目录名）来容纳长文件名，这样就透明地扩展了基础FAT文件系统。为了支持老系统，每个文件和目录都可以用一个特定的符合8.3规范的DOS别名来访问。比如名为*Downloaded Program Files*的目录可以用*DOWNLO~1*这样的名字来访问。

VFAT和Unix文件系统有四个明显的差异：

- FAT文件系统不区分大小写。在Unix中，使用错误的大小写访问文件（比如用*MYFAVORITEFILE*访问*myfavoritefile*文件）通常会失败，但对于FAT或VFAT来说，这并不会带来任何问题。
- FAT并没有使用斜线，而是使用反斜线来作为路径分隔符。这对Perl开发人员来说是件麻烦事，因为反斜线本身是Perl的转义字符。对于那些单引号中的路径来说（比如`$path = '\dir\dir\filename'`）这还算小问题，而如果要表达连续的多个反斜线（比如`\\server\dir\file`）就会有很多问题。因为这样就必须很小心地对多个反斜线本身进行反斜线转义。有些Perl函数和模块能接受反斜线开头的路径，但最好不要过分依赖它们。最好的办法还是硬着头皮写`\\\\winnt\\temp\\`这样的代码，别总指望别人替你转换。
- 在FAT文件系统中，文件和目录能设定特殊的标志，称为属性（attribute），例如“只读”或者“系统”。
- FAT文件系统的根是开始于所在磁盘的盘符，如某个文件的绝对路径可能是`c:\home\cindy\docs\resume\current.doc`。

FAT32和NTFS文件系统与VFAT文件系统有相似的语义设计。对长文件名的支持和文件系统根目录的表达方式都相同。然而NTFS在Unicode方面也提供了支持，这使得它的设计更加复杂。Unicode是一种支持地球上各种语言中所有字符的多字节编码方案（scheme）。

NTFS还有一些区别于其他Windows和基础Unix文件系统的地方。在本章的后面，我们会用代码展示一些具体细节，比如文件系统限额。NTFS支持ACL，这让文件和目录的访问控制得到更好的支持。它还支持一些我们不会提及的特殊功能，比如文件加密和压缩。另外请注意的是，Vista只能安装在用NTFS格式化的文件系统上。

在进入下一个操作系统之前，请先容我介绍通用命名约定（universal naming convention, UNC）。UNC是在网络环境中访问文件和目录的便利方式。在UNC名中，盘符和冒号都被替换为`\\server\sharename`。这个约定也遭受我们刚才看到的相同Perl反斜线语法冲突之苦，因此像下面这样的一串斜牙签是屡见不鲜的：

```
$path = '\\\\server\\sharename\\directory\\file';
```

Mac OS X

在写这本书第一版的时候，OS X才刚刚进入大众的视野。经典的Mac OS采用一种名为Mac OS Hierarchical File System的文件系统，或者简称为HFS。这个文件系统与之前讨论过的文件系统都非常不同，它有非常不同的文件语义并且需要特别的支持才能供Perl访问。Mac OS 8.1引入了HFS的衍生版本，名为HFS+，这成了OS X的默认文件系统。

^[注2] 新版本的OS X在持续改进这个文件系统的功能。

虽然花了不少时间，也经历了几个版本才把HFS+文件系统改进到现在的样子，但其实目前的HFS+文件系统语义在使用上非常类似于Unix的文件系统。文件和路径都是用相同的方式来指定，甚至HFS+还能支持BSD扩展属性（比如ACL）。只要你使用标准Perl机制来与文件系统交互，就能使程序在HFS+和其他Unix文件系统上运行。

注意： 如果你很好奇，想尝试通过非标准方式来访问HFS+文件系统，也可以考虑采用以下方法：

- 直接调用Mac OS X命令行工具，比如使用`chmod +a...`，只要已经使用`fsaclctl`开启了ACL控制。
- 使用Dan Kogai的`MacOSX::File`模块。这个模块还支持你访问一些“过时”的扩展属性（比如`type`、`creator`等），这些属性在OS X之前的文件系统中曾经非常重要。

标准UFS与HFS+文件系统有一个关键的区别。默认情况下，^[注3] HFS+并不会区分大小写：对它来说BillyJoeBob和billyJoebob是同样的文件，也就是说如果你对第一个调用`open()`，而第二个才是正确的文件名，最终你还是能得到指向正确数据的文件句柄。对于Perl程序来说，你并没有任何需要处理的地方，只需要记住这个例外就好。主要的问题是删除文件，因为这可能会导致意外丢失文件。

文件系统差异汇总

表2-1总结了以上讨论的所有差异，还外加了几条有意思的差别。

注2： 顺便说一下，其实你可以在OS X下创建UFS格式化的文件系统，而且在写作这本书的时候，完全的ZFS支持也很快就要发布了。

注3： 其实目前的OS X版本可以创建区分大小写的HFS+卷，但是这样做往往会有危险。区分大小写会导致应用程序崩溃，而崩溃的程序并没有用特殊的方法访问文件系统。所以如果没有什么好的理由，请别这么做。

表2-1：文件系统对比

操作系统和文件系统	路径分隔符	文件名长度规范	绝对路径格式	相对路径格式	特有功能
Unix (Berkeley 快速文件系统及其他)	/	字符数量随操作系统而定	/dir/file	dir/file	随各种操作系统而加入的改进
Mac OS (HFS+)	/	255个Unicode 字符	/dir/file	dir/file	Mac OS传统的文件属性 (如creator/type) ,BSD扩展属性
基于Windows的操作系统 (NTFS)	\	255个Unicode 字符	Drive:\dir\file	dir\file	文件加密和压缩
DOS (基础FAT)	\	8.3	Drive:\dir\file	dir\file	属性

使用Perl处理文件系统差异

Perl能帮我们写出遵从上述文件系统差异的代码。因为自带的File::Spec模块能隐藏文件系统的差异。例如，我们可以传递路径组件给catfile方法：

```
use File::Spec;

my $path = File::Spec->catfile(qw{home cindy docs resume.doc});
```

\$path在Windows系统中被设置为home\cindy\docs\resume.doc，而在Unix或OS X中则被设置为home/cindy/docs/resume.doc。File::Spec还有类似curdir和updir这样的函数，它们用来返回代表当前目录和父目录的标记符号（“.”和“..”）。这个模块提供的方法能够帮助程序更加抽象地构造并处理符合路径规范的文件名。如果你不喜欢用面向对象的风格来写程序，那么File::Spec::Functions模块能提供File::Spec中的所有方法。

如果你觉得File::Spec的接口设计得有些特殊（如catfile()这个名字只是对那些熟悉Unix cat命令的人来说才容易理解，因为它的工作就是一种连接（concatenate）），那么可以选择Ken Williams的Path::Class模块。然而它不像File::Spec那样随Perl发行，而是需要独立安装，不过你会觉得安装很值得。以下是使用方法：

首先使用指定路径组件的自然语法创建一个Path::Class::File或者Path::Class::Dir对象：

```
use Path::Class;
```

```
my $pcfile = file(qw{home cindy docs resume.doc});
my $pcdir = dir(qw{home cindy docs});
```

现在\$pcfile和\$pcdir这两个对象已经可用了。这看上去有些神奇：只要像使用其他任何标量那样使用它们就可以获得匹配当前操作系统的路径。比如：

```
print $pcfile;
print $pcdir;
```

这会输出home/cindy/docs/resume.doc和home/cindy/docs，或者home\cindy\docs\resume.doc和home\cindy\docs，类似之前File::Spec所做的那样。

尽管\$pcfile和\$pcdir能转换成路径字符串，但它们实际上是对象。因此有很多方法可以调用它们。这些方法中包含了File::Spec模块中的方法和一些扩展。比如：

```
my $absfile = $pcfile->absolute;    # 返回 $pcfile 的绝对路径
my @contents = $pcfile->slurp;      # 一次读入该文件的所有内容
$pcfile->remove();                  # 实际删除该文件
```

Path::Class还有两个技巧值得一提，第一个是解析现有路径：

```
use Path::Class;

# 它希望接收的是完整路径，而不是路径的组件
my $pcfile = file('/home/cindy/docs/resume.doc');

print $pcfile->dir();              # 注意：这会返回一个 Path::Class::Dir
                                  # 而它在此被转化成了字符串
print $pcfile->parent();            # 和 dir() 等效，不过读起来更加顺畅
print $pcfile->basename();         # 单纯的文件名，不包含路径
```

另外一个技巧是可以写出在某个平台运行，却能理解另一个平台的文件路径的代码。例如想要写出一个在Linux平台上运行的Web应用程序，用来指导用户搜索其他本地Windows机器上的文件。这时，可以用Path::Class思考其他操作系统的语义，你需要显式导出另外两个方法：foreign_file()和foreign_dir()。这两个方法以操作系统类型作为第一个参数：

```
use Path::Class qw(foreign_file foreign_dir);

my $fpcfile = foreign_file('Win32', qw{home cindy docs resume.doc});
my $fpcdir = foreign_dir('Win32', qw{home cindy});
```

现在\$fpcfile会输出home\cindy\docs\resume.doc，哪怕代码在Mac上运行也是一样。这种需求可能并不常见，但在某些场合却非常有用。

手动遍历文件系统

到目前为止我们还没谈到实际的 Perl 应用，所以你可能已经有些手痒了。我们先尝试进行文件系统遍历，这可能是关于文件系统最基础的系统管理任务了。这个典型的任务包括搜索系统中所有的目录，并对某些特定的文件执行某项操作。每个操作系统都有为此设计的专门工具：Unix 下的 `find` 命令、Windows 下的 Search、Mac OS 的 Spotlight 或者 Finder 中的搜索框（在 Terminal 中则是 `find` 命令）。以上这些工具都能完成搜索，但是往往缺少在搜索的同时执行特殊操作的能力。这一节就展示 Perl 是如何来让你编写出复杂的文件遍历代码，我们从基础开始，逐渐增加难度。

好的，开始之前先假设我们有如下问题如要解决：我们是 Unix 系统管理员（先假定是 Unix，其他系统慢慢来），文件系统正要被撑满，而我们没有买磁盘的预算。

因为不可能加磁盘，所以我们只能更好地利用现有的资源。第一步就是尝试删除文件系统中不再有用的文件。在 Unix 中，最好的删除候选文件就是程序崩溃导致的 core 文件。用户往往没有注意它们或懒得删除它们，但这些文件会消耗大量的磁盘空间，而且没有任何用处。所以我们的程序要在文件系统中找出并删除它们。

要手动遍历文件系统，首先需要读取某个目录的内容，并且完成下一个目录的定位。我们可以简化问题，从当前目录开始，看看其中是否有 core 文件或者需要被搜索的其他目录。

首先，我们用类似打开文件的方法打开目录。如果打开失败，我们的程序就会退出并且打印由 `opendir()` 调用 (\$?) 设置的错误消息：

```
opendir my $DIR, '.' or die "Can't open the current directory: $!\n";
```

这会给我们一个目录句柄 `$DIR`。我们可以将它传递给 `readdir()` 以获得当前目录中的文件和目录列表。如果 `readdir()` 不能读取这个目录，我们的代码会打印错误消息（解释原因）并且退出程序：

```
# 把此目录下的文件名和目录名都读入@names
my @names = readdir $DIR or die "Unable to read current dir:$!\n";
```

然后关闭打开的目录句柄：

```
closedir $DIR;
```

现在可以对这些名字进行操作：

```
foreach my $name (@names) {
    next if ($name eq '.'); # 跳过当前目录条目
    next if ($name eq '..'); # 跳过父目录条目
```



```

if (-d $name) {          # 这是一个目录吗?
    print "found a directory: $name\n";
    next;                # 可以继续处理下一个条目
}
if ($name eq 'core') {   # 这是一个名叫“core”的文件吗?
    print "found one!\n";
}
}

```

到此为止，这些简单的代码已经可以处理单个目录了，但还不能在文件系统中移动，更别提遍历了。要想遍历文件系统，我们得挨个进入找到的子目录，并且检查其中的内容。如果子目录中还有子目录，我们也得一并检查。

每当我们需要对某个容器和其中的所有子容器执行同样的操作的时候，往往需要递归的解决方案（在科学计算中用得更多）。只要递归的层次不深，并且能确保不要回到原地（也就是说每个容器的子容器都是属于它的直接上级，不会被其他容器包含），这样的解决方案常常最有效。这样我们可以扫描某个目录和它的子目录、孙目录等。

如果你是第一次看到递归代码（也就是自己调用自己的代码），可能会有些惊讶。写递归代码有点像绘制俄罗斯套娃，最大的娃娃套着稍小点的娃娃，一直这样直到最里面放着一个小到不能再小的娃娃。

你可以按以下步骤来绘制这样的娃娃：

1. 检查手上的娃娃，看看其中是否有更小的娃娃。如果有，就打开并拿出其中的娃娃。
2. 重复第一步，直到拿到最小的娃娃。
3. 在娃娃上面绘图，颜料干后把它放回之前的那个娃娃中去。
4. 重复第3步，直到你把所有的娃娃都画好。

这个流程一直重复相同的步骤。如果手上的事情还有嵌套的任务，则把手上的事情放一放，先去做嵌套的任务。如果手上的事情没有嵌套任务，则直接处理事情，然后回到最近一次放下的事情那儿，一直这样直到做完所有事情。

用编程的术语来讲，这通常需要使用子例程（subroutine）来处理容器。子例程先检查当前容器是否有子容器，如果有则一直调用自己来处理所有子容器。如果没有子容器，则执行一些操作并返回调用它的代码。如果你对调用自己的代码不熟悉，我建议你坐下来准备好纸和笔，画图追踪程序流转，直到你已经确实弄明白了程序的运作方式。

让我们先来看递归代码。为了让我们的代码能递归调用，首先把扫描目录并对其内容起作用的操作封装成子例程，可以命名为ScanDirectory()。这个子例程有一个调用参数，也就是需要扫描的目录。扫描完毕后，子程序会返回到被调用的地方。下面是更新

之后的代码：

```
#!/usr/bin/perl -s

# 请注意这里使用了-s用于开关（switch）处理。如果在Windows下缺少perl文件的关联
# 你必须使用perl -s script这样的方式来调用这个脚本。
# 使用-s这样的选项有时也被认为是“复古”——
# 大多数程序员会加载Getopt::系列模块来完成开关解析。

use Cwd; # 用于定位当前工作目录的模块

# 这个子例程接受某个目录名，并且递归搜索此目录的所有子目录
# 从而发现其中所有名为“core”的文件
sub ScanDirectory {
    my $workdir = shift;

    my $startdir = cwd;    # 开始前先保存当前目录

    chdir $workdir or die "Unable to enter dir $workdir: $!\n";
    opendir my $DIR, '.' or die "Unable to open $workdir: $!\n";
    my @names = readdir $DIR or die "Unable to read $workdir: $!\n";
    closedir $DIR;

    foreach my $name (@names) {
        next if ( $name eq '.' );
        next if ( $name eq '..' );

        if ( -d $name ) {    # 这是一个目录吗？
            ScanDirectory($name);
            next;
        }
        if ( $name eq 'core' ) {    # 这是名为“core”的文件吗？
            # 如果在启动时设置了-r选项，那么执行删除操作
            if ( defined $r ) {
                unlink $name or die "Unable to delete $name: $!\n";
            }
            else {
                print "found one in $workdir!\n";
            }
        }
    }
    chdir $startdir or die "Unable to change to dir $startdir: $!\n";
}

ScanDirectory('.');
```

上述程序最重要的变化在于找到子目录后的行为，现在程序不再打印“found a directory!”，而是递归调用自己来查看子目录。一旦那个子目录完全遍历一次之后（也就是说这个ScanDirectory()调用返回之后），程序就可以接着查看当前目录中的其他内容。

为了让代码能真正实现删除core文件的功能，我们也一并加入了文件删除的功能。请注意代码是如何工作的：只有在命令行开关-r（也就是“remove”）启用的时候，程序才会真的删除文件。

我们在这里使用Perl内置的-s开关来完成命令行选项的自动解析。所以第一行代码写成#!/usr/bin/perl -s。这是解析命令行选项的最简单方式^[注4]，要想处理更复杂的命令行选项，请使用Getopt::系列模块。如果代码调用的时候加上了-r这样的命令行开关，则会自动创建一个\$r这样的全局变量。在我们的代码中，如果调用Perl时没有-r，我们就会简单打印找到了core文件的信息。

警告：在编写自动化工具的时候，切记要把毁灭性的操作设计得难以完成。请记住此忠告，Perl的功能强大，因此也极有可能伤害到文件系统，而且往往不必费力就能做到。

好了，到现在为止那些以Windows为主的读者可能还在想这个例子并不实用，让我告诉你怎么把它变得有用。只要把下面这行代码：

```
if ($name eq 'core') {
```

改成：

```
if ($name eq 'MSCREATE.DIR') {
```

就能有效地删除那些让人厌烦的Microsoft安装程序遗留的空文件。虽然这些文件如今越来越罕见了，但这种类型的文件肯定会继续出现。

好了，有了这个工具之后让我们回到本章开头的那个问题。笔记本摔到地上之后，我非常需要一个工具来判断磁盘上哪些文件可以读出，而且要跳过那些损坏的文件。

这是我实际用过的代码（最起码非常相似）：

```
use Cwd; # 用于定位当前工作目录的模块
$|=1;   # 关闭I/O缓冲机制

sub ScanDirectory {
    my $workdir = shift;

    my $startdir = cwd;    # 开始前先保存当前目录

    chdir $workdir or die "Unable to enter dir $workdir: $!\n";

    opendir my $DIR, '.' or die "Unable to open $workdir: $!\n";
    my @names = readdir $DIR;
    closedir $DIR;
```

注4：默认情况下，-s在和use strict一起使用时会有些问题，所以在正式脚本中请避免使用它。

```

foreach my $name (@names) {
    next if ( $name eq '.' );
    next if ( $name eq '..' );

    if ( -d $name ) {      # 这是目录吗?
        ScanDirectory($name);
        next;
    }
    CheckFile($name)
or print cwd . '/' . $name . "\n";    # 打印坏文件名
}

chdir $startdir or die "Unable to change to dir $startdir:!\n";
}

sub CheckFile {
    my $name = shift;
    print STDERR 'Scanning ' . cwd . '/' . $name . "\n";
    # 尝试读取此文件的目录信息
    my @stat = stat($name);
    if ( !$stat[4] && !$stat[5] && !$stat[6] && !$stat[7] && !$stat[8] ) {
        return 0;
    }
    # 尝试打开此文件
    open my $T, '<', "$name" or return 0;

    # 一次一个字节地读取文件, 把读出的数据存入$discard
    for ( my $i = 0; $i < $stat[7]; $i++ ) {
        my $r = sysread( $T, $discard, 1 );
        if ( $r != 1 ) {
            close $T;
            return 0;
        }
    }
    close $T;
    return 1;
}

ScanDirectory('.');

```

这段代码和之前的代码的区别在于加入了一个用于检查遇到的文件的子例程。我们尝试用stat()函数来检查每个文件的目录信息(比如文件大小)是否可读。如果这个信息不可读,那么文件已经损坏;如果能够读取目录信息,我们会尝试打开文件。并且为了判断文件是否真正完好,我们尝试一次一个字节地读取文件。当然,这么做并不能保证文件没有损坏(也许读出的内容并不正确),但这起码能保证文件可读。

你也许会有疑问:为什么这段代码要使用sysread()这样罕见的函数来读取文件呢?虽然大多数人习惯于使用Perl的<>或read()来读取文件,但是sysread()提供了可以绕过常规的读取缓冲的能力。如果已经知道文件在X位置损坏了,还尝试着读取X+1、X+2和X+3这些位置就没有意义了(而且浪费时间),我们希望代码能在这种情况下立刻停

止继续对文件的读取。通常磁盘读取缓冲能提高响应速度，但对于那些损坏了的文件来说，这只能导致磁头不断发出难听的声音。

好的，现在你已经看到了我的代码，让我来告诉你故事的结局。在这个脚本运行了几乎整个晚上之后，它报告了16 000个文件中的95个文件已损坏。而且幸运的是，这本书用到的所有文件无一损坏，我把完好的文件备份到另一台机器上继续工作；Perl挽救了大局。

使用File::Find模块来遍历文件系统

好了，我们已经展示了如何实现基础的文件系统遍历，现在是时候介绍更快也更酷的解决方案了。Perl自带了File::Find模块用来模拟Unix的find命令。使用这个模块的最简单的方法是用find2perl命令来生成原型Perl代码。

比如你需要代码来搜索/home目录中存放的名为beesknees的文件。使用Unix的find命令的命令行需要如此写：

```
% find /home -name beesknees -print
```

把同样的选项提供给find2perl：

```
% find2perl /home -name beesknees -print
```

以下是生成的代码：

```
#!/usr/bin/perl -w
eval 'exec /usr/bin/perl -S $0 ${1+"$@"}'
    if 0; # $running_under_some_shell

use strict;
use File::Find ();

# 如果使用 AFS，请设置 $File::Find::dont_use_nlink 变量，
# 因为 AFS 在符号链接的处理上非常特殊。

# 为了 &wanted 调用的便利，引入 -eval 语句：
use vars qw/*name *dir *prune*/;
*name  = *File::Find::name;
*dir   = *File::Find::dir;
*prune = *File::Find::prune;

sub wanted;

# 遍历指定的文件系统
File::Find::find({wanted => \&wanted}, '/home');
exit;

sub wanted {
    /^beesknees/z/s &&
```

```
    print("$name\n");
}
```

*find2perl*生成的代码非常容易阅读。它会载入必要的File::File模块，设置一些便于使用的变量（之后会仔细分析它们），使用“wanted”子例程和起始目录的名称作为参数来调用File::Find::find，我们稍后分析子例程的功能，顺便展示很多有趣的编程技巧。

我们先介绍一些代码之外的背景信息，然后开始尝试修改这段代码。

- 创建File::Find模块的人花了很多力气来确保这个模块能跨平台工作。通过这个模块的内部例程来确保同样的Perl代码能在Unix、Mac OS X、Windows和VMS上运行。
- *find2perl*生成的代码包含过时的use vars编译指令，这个语法在Perl 5.6中已经被our()函数代替了。我感觉保留这些老代码是为了向前兼容。指出这个是为了避免读者意外学习过时的语法。

现在让我们来看看将要改进的wanted()子例程。在遍历过程中，File::Find::find()会为遇到的每个文件或目录以当前文件名或者目录名调用wanted()子例程，而且只调用一次。选择合适的文件或目录并对它们进行操作是wanted()子例程的职责。在之前的样本代码中，它会检查文件名或目录名是否匹配字符串beesknees。如果匹配，则执行&&操作符右面的print语句来打印找到的文件或目录的名称。

另外还要指出两个实际编程要点，这样能保证我们的wanted()子例程足够健壮。首先因为wanted()会被每个文件或目录调用一次，所以要保证代码短小和高效。越是能及早退出File::Find::find()，例程处理下一个文件或目录的速度就越快，这会加速整体程序的运行。另外需要记住的一点是，我们之前刚刚提过的关于幕后细节的可移植性。如果在File::Find::find()运行过程中需要调用特定于某种操作系统的wanted()的话，简直叫人难为情，当然除非碰到完全无法避免的情况。不妨打开File::Find模块，看看它的源代码，并读一读perlport说明文档，或许会得到一些避免此类问题发生的启发。

这是第一次使用File::Find，让我们先尝试用它来重写之前的core文件删除程序，然后稍加改进。首先让我们输入：

```
% find2perl -name core -print
```

这会输出（节选）：

```
use strict;
use File::Find ();

use vars qw/*name *dir *prune/;
*name  = *File::Find::name;
*dir   = *File::Find::dir;
*prune = *File::Find::prune;
```

```
File::Find::find({wanted => \&wanted}, '.');

sub wanted {
    /^core\z/s &&
    print("$name\n");
}
```

然后我们给Perl调用行加上-s开关，然后修改wanted()子例程：

```
my $r;
sub wanted {
    /^core$/ && print("$name\n") && defined $r && unlink($name);
}
```

这样就可以实现我们期望的删除功能，而且是通过命令行开关-r启动的。下面的小技巧还能让我们的代码在删除时更加谨慎：

```
my $r;
sub wanted {
    /^core$/ && -s $name && print("$name\n") &&
    defined $r && unlink($name);
}
```

改进之后的程序会检查任何名为core的文件的大小，确定它不是零字节的文件后才会打印文件名（或干脆删除）。有经验的用户往往会在他们的home目录下创建一个指向/dev/null的软链接，名叫core，因为这样可以避免核心转储（core dump）出现在他们的目录中。这里的-s检查避免了意外删除链接和零字节文件。如果要更进一步，我们还可以做两个额外的检查：

1. 打开并检查文件，确认它确实是个core文件，既可以用Perl来打开文件，也可以通过调用Unix的file命令判断。当然要注意，如果在网络文件系统中判断可能会出问题，因为core文件可能来自不同架构的主机。
2. 检查文件的修改日期。如果某人正在用它调试程序，那么他一定不希望core文件从眼皮底下消失。

在进一步深入检查代码之前，可以先解释一下那几行看来比较神秘的变量别名设定代码：

```
*name    = *File::Find::name;
*dir     = *File::Find::dir;
*prune   = *File::Find::prune;
```

Find::File构造了一系列变量供wanted()子例程使用，其中比较重要的那些列在表2-2中。

表2-2: File::Find变量

变量名	含义
<code>\$_</code>	当前文件名
<code>\$File::Find::dir</code>	当前目录名
<code>\$File::Find::name</code>	当前文件名的完全路径 (比如: <code>"\$File::Find::dir/\$_"</code>)

这些变量会在我们下面的代码中出现。

让我们先暂时离开Unix世界一会儿,看看Windows相关的例子。我们可以稍微修改一下代码,让它搜索当前磁盘上文件系统中的隐藏文件(也就是设定了HIDDEN属性的文件)。这个例子可以在NTFS和FAT文件系统上工作:

```
use File::Find ();
use Win32::File;

File::Find::find( { wanted => \&wanted }, '\\' );

my $attr; # 在此进行全局定义,而不是在wanted()中定义
          # 这样可以避免每次调用时反复定义局部的$attr变量

.sub wanted {
    -f $_
    && ( Win32::File::GetAttributes( $_, $attr ) )
    && ( $attr & HIDDEN )
    && print "$File::Find::name\n";
}
```

下面是一个与NTFS相关的范例,用来查找那些设置了允许*Everyone*这个组的成员进行所有操作(Full Access)的文件并打印它们的名字:

```
use File::Find;
use Win32::FileSecurity;

# 查找 Full Access 对应的 DACL 掩码
my $fullmask = Win32::FileSecurity::MakeMask(qw(FULL));

File::Find::find( { wanted => \&wanted }, '\\' );

sub wanted {
    # 这次我们需要在每次调用时获得新的 %users
    my %users;

    ( -f $_ )
    && eval {Win32::FileSecurity::Get( $_, \%users )}
    && ( defined $users{'Everyone'} )
    && ( $users{'Everyone'} == $fullmask )
    && print "$File::Find::name\n";
}
```


在这段代码中，我们查询了每个文件的访问控制列表，检查是否有关于`Everyone`组的条目。如果有，检查它的设定是否为Full Access（通过`MakeMask()`函数来计算），最终打印出被找出文件的绝对路径。

注意：你可能会好奇为什么要用`eval()`来调用之前的那段代码。虽然说明文档中说明了`Win32::FileSecurity`能正常地在`$!`中设置错误信息，其实在特殊情况下它可能会让人惊讶地崩溃掉。这在文档中被列为bug，不过很容易被忽视。

不幸的是，有两种情况会导致这个模块产生不良反应：首先是它无法读取的内存分页文件，其次是空DACL（一种特殊设置的ACL权限）。所以我们用`eval()`捕捉并忽略这样的反常行为。

顺便说一下，操作系统中的某些组件（比如说Explorer）也认为空DACL会导致Everyone组的访问权限问题。如果我们想要显示这些特殊的文件，需要检查`$@`。

另外还有一个实用的例子，证明了这些简单的代码会有多大用处。几个月之前，在我尝试对笔记本电脑的一个NTFS分区进行碎片整理的操作时，所使用的软件汇报了“Metadata Corruption Error”。我在软件供应商的站点上搜索到这个错误，它是这么描述的：“这个问题的原因在于某个文件的文件名过长，已经超出了Windows NT能够支持的长度”。后面跟着解决方案，建议将所有文件夹拷贝到其他地方，然后仔细比较两个地方的所有文件，那些没有拷贝过去的文件就是有问题的文件。

这个建议看起来非常可笑，因为这样的分区容纳了太多的文件，而且拷贝也会消耗太多的时间。所以我花了大概一分钟的时间写出了下面的代码（已经改掉了一些旧的代码）来尝试这个建议：

```
use File::Find;

my $max;
my $maxlength;

File::Find::find( { wanted => \&wanted }, '.' );

print "max:$max\n";

sub wanted {
    return unless -f $_;
    if ( length($_) > $maxlength ) {
        $max = $File::Find::name;
        $maxlength = length($_);
    }
    if ( length($File::Find::name) > 200 ) { print $File::Find::name, "\n"; }
}
```

这段代码会打印文件名最长的文件及所有文件名长于200个字符的文件。任务就此完成，感谢Perl。

什么时候不建议使用File::Find模块

什么情况下File::Find这个方法不太好用呢？有下面三种情况：

1. 如果想要遍历的文件系统已经不太正常，请不要使用它。比如之前我们修复摔坏的笔记本电脑中的文件，那个例子中的文件系统已经出现了问题，即使在空目录中也没有“.”或“..”目录。这会让File::Find模块晕头转向。
2. 如果你的代码会在遍历文件系统的过程中修改文件名或者目录名，那么File::Find会做出让人讨厌的不可预测的行为。
3. 如果你想要遍历挂载在系统上的非本地的文件系统（比如Windows机器上挂载的Unix文件系统），那么File::Find会尝试用本地操作系统的文件系统语义来访问远程文件系统。

遇到这些情况的概率不大，但如果发生这样的事情，请使用我们之前介绍过的方法来手动遍历文件系统。

现在让我们回到Unix世界，并介绍一个更复杂的案例来结束这一章。在系统管理的圈子里面，常常被忽略的（并值得提倡的）就是提供帮助，使用户能自行完成任务。如果用户通过使用你的工具来解决他们自己的问题，那么就能实现双赢。

这一章大多数的篇幅都在讨论如何解决文件系统满载的问题。这个问题的根源往往是用户不明白他们的系统，也有可能是因为磁盘管理工作太繁琐了。很多的系统支持请求的第一句话就是“我的home目录满了，但是我不知道为什么。”下面就是一个名为*needspace*的脚本的快速原型，它应该能帮助用户找到问题所在。用户需要做的只是键入*needspace*，然后脚本就会列出用户的home目录中可以删除的东西。它会尝试搜索两类文件，一种是core或者备份文件，另一种是可以自动重建的文件。让我们先看看代码：

```
use File::Find;
use File::Basename;
use strict;

# 原文件扩展名和衍生文件扩展名的哈希
my %derivations = (
    '.dvi' => '.tex',
    '.aux' => '.tex',
    '.toc' => '.tex',
    '.o'   => '.c',
);

my %types = (
    'emacs' => 'emacs backup files',
    'tex'   => 'files that can be recreated by running La/Tex',
```

```

'doto' => 'files that can be recreated by recompiling source',
);

my $targets;      # 搜索的结果会存放在这个哈希的哈希中
my %baseseen;     # 用来记录访问过的原文件

```

我们先加载需要的模块，首先是一直以来的老伙伴`File::Find`，另外还有`File::Basename`。这个新模块非常有用，因为它能分析路径名。我们还要初始化一个文件扩展名的哈希表，例如我们知道运行TeX或者LaTeX命令可以从`happy.tex`生成`happy.dvi`，也可以在`happy.c`上进行C语言编译产生`happy.o`。我说“可以”是因为有时候多个文件配合才能产生一个文件。我们目前只是针对文件扩展名做简单的猜测，真正意义上的依赖分析实在太复杂了，我们现在无暇顾及。

然后，我们通过用户的ID来定位他们的home目录，执行脚本的用户的ID可以通过`$<`变量来得到，然后把它作为参数传递给`getpwuid()`即可。这个函数会返回一组关于用户的信息，更多的内容请参考第3章。在这个数组中，数组索引[7]可以用来获取用户的home目录名。当然也可以通过shell的方式来获取这个信息（比如检查`$HOME`环境变量），但为了代码的可移植性，我们先不用它。

一旦定位到home目录，我们就用`find()`进行搜索，类似之前的搜索方式：

```

my $homedir = ( getpwuid($<) )[7];    # 找到用户的 home 目录

chdir($homedir)
    or die "Unable to change to your homedir $homedir:$!\n";

$| = 1;                                # 以无缓冲方式打印标准输出

print 'Scanning';
find( \&wanted, '.' );
print "done.\n";

```

这里是被调用的`wanted()`子例程。它先查找`core`文件和`emacs`备份与自动保存文件。我们假定这些文件是可以删除的（不必检查原始文件），但请注意这个假定可能会有风险。如果这类文件存在，那么我们把文件的大小和路径存放在一个哈希的哈希中去，内部哈希的键是文件的路径，而值则是文件的大小。

下面检查可自动生成文件的代码也很类似。它调用一个名为`BaseFileExists()`的例程来检查指定文件是否可以通过同一目录下的另一个文件来生成。如果这个例程返回`true`，那么我们就把此文件的文件名和文件大小信息存储起来供后续处理：

```

sub wanted {

    # 为每一个处理中的目录打印一个点号，这样用户可以看到程序还在运行
    print '.' if ( -d $_ );

```

```

# 我们只关心文件
return unless -f $_;

# 检查 core 文件
$_ eq 'core'
    && ( $targets->{core}->{$File::Find::name} = ( stat($_) )[7] )
    && return;

# 检查 emacs 备份和自动保存文件
( /^#.*$/ || /^$/ )
    && ( $targets->{emacs}->{$File::Find::name} = ( stat($_) )[7] )
    && return;

# 检查可删除的tex衍生文件
( /\.dvi$/ || /\.aux$/ || /\.toc$/ )
    && BaseFileExists($File::Find::name)
    && ( $targets->{tex}->{$File::Find::name} = ( stat($_) )[7] )
    && return;

# 检查可删除的.o衍生文件
 /\.o$/
    && BaseFileExists($File::Find::name)
    && ( $targets->{doto}->{$File::Find::name} = ( stat($_) )[7] )
    && return;
}

```

下面是检查一个文件是否衍生自同一目录下另一个基文件（base file）（好比*happy.o*依赖于*happy.c*那样）的例程：

```

sub BaseFileExists {
    my ( $name, $path, $suffix ) = File::Basename::fileparse( $_[0], '\..*' );

    # 如果我们不知道此文件是从哪个文件衍生的
    return 0 unless ( defined $derivations{$suffix} );

    # 如果我们曾经记录过此文件的基文件
    return 1
        if ( defined $baseseen{ $path . $name . $derivations{$suffix} } );

    # 如果文件（或者通过链接来引用的文件）存在而且大小不为零
    # 在我们缓存信息后返回成功
    return 1
        if ( -s $name . $derivations{$suffix}
            && ++$baseseen{ $path . $name . $derivations{$suffix} } );
}

```

这是代码执行的逻辑：

1. `File::Basename::fileparse()`用来将路径分隔为文件名、引导路径和后缀（比如 *resume.dvi*、*/home/cindy/docs/*和*.dvi*）。

2. 首先检查后缀是否属于可以被自动生成的那种。如果不是，就直接返回0（而它其实是标量上下文的“false”）。
3. 然后再检查是否看到过这个文件的“基文件”，如果发现的话就返回true。有时一个基文件可以产生很多个衍生文件，比如TeX/LaTeX。我们这里为了节约资源只做最容易的文件衍生检查。
4. 如果之前没有看到它的基文件，我们就去检查基文件是否存在。如果基文件存在，而且大小不是0字节，那么我们就记录基文件的信息并且返回1（而它其实是标量上下文的“true”）。

剩下的工作就是把遍历文件系统期间采集到的信息打印出来：

```
foreach my $path ( keys %{ $targets->{core} } ) {
    print 'Found a core file taking up '
        . BytesToMeg( $targets->{core}{$path} )
        . 'MB in '
        . File::Basename::dirname($path) . ".\n";
}

foreach my $kind ( sort keys %types ) {
    ReportDerivFiles( $kind, $types{$kind} );
}

sub ReportDerivFiles {
    my $kind    = shift;          # 正在报告的文件类型
    my $message = shift;          # 此类文件的描述信息
    my $tempsize = 0;

    return unless exists $targets->{$kind};

    print "\nThe following are most likely $message:\n";

    foreach my $path ( keys %{ $targets->{$kind} } ) {
        $tempsize += $targets->{$kind}{$path};
        $path =~ s|^\.\/|~/|;      # 修改路径，使得输出更加美观
        print "$path ($targets->{$kind}{$path} bytes)\n";
    }
    print 'These files take up ' . BytesToMeg($tempsize) . "MB total.\n\n";
}

sub BytesToMeg {
    # 把字节数转化成 ×.××MB 格式
    return sprintf( "%.2f", ( $_[0] / 1024000 ) );
}
```

在结束这一节之前，我必须说明，这个例子还值得进一步扩充，因为还有很多可能的应用，比如：

- 搜索Web浏览器的cache目录（那里是磁盘空间清理程序的最爱）。

- 提供删除垃圾文件的选项。可以使用`unlink()`函数和`File::Path`模块的`rmpath`子例程来完成。
- 对文件执行更深入的分析，而不只是基于文件名进行猜测。

使用File::Find::Rule模块来遍历文件系统

`File::Find`模块提供了一种快捷遍历文件系统的方法。它是随Perl一同发行的，不必自己动手安装。但是如果你写过很多基于`File::Find`的脚本，应该会意识到你正在不断重复着某些代码。这时候你可能想找到某种方法来避免一直重复下去，因为一再套用自己的代码模板也不能算是好办法。如果你乐意考虑那些需要安装的Perl模块，我会向你隆重推荐`File::Find::Rule`。

`File::Find::Rule`是一个（其实已经是一个系列，稍后就会看到）非常棒的模块，由Richard Clamp创作。这个模块提供了两个比`File::Find`更灵巧的接口。如果你不想继续忍受`File::Find`，我强烈推荐你看看`File::Find::Rule`模块。让我们先看看它的优点。

首先要说的是，Clamp的模块让编写收集来自文件的文件或目录列表的脚本变得更加容易了。在使用`File::Find`时，你必须自己完成筛选和记录的任务，而`File::Find::Rule`可以帮你完成这些事情。你可以简单地给它一个遍历的开始位置以及一系列用于筛选文件的方法，这个模块就能在遍历结束后产生一个文件列表，也可以产生一个遍历列表成员的方法。我们先尝试最简单的表达式：

```
use File::Find::Rule;
my @files_or_dirs = File::Find::Rule->in('.');
```

现在`@files_or_dirs`包含了（当前目录及所有子目录中的）所有文件和目录，正如`in()`方法描述的那样。如果我们只想找文件而不关心目录，可以再加上`file()`：

```
my @files = File::Find::Rule->file()->in('.');
```

如果我们只想找出名字结尾是`.pl`（也就是和Perl相关的那些）的文件：

```
my @perl_files = File::Find::Rule->file()->name('*.pl')->in('.');
```

如此这般，我们可以继续在方法链中加入更多的方法，使它实际上担当过滤器的角色。`File::Find::Rule`还有一个过程化的接口，这样如果你不太喜欢面向对象的语法，还可以把代码改写成这样：

```
my @perl_files = find( file => name => '*.pl', in => '.' );
```

我自己不觉得这样的写法更加易读，但是没准有人会喜欢这样写。

在进一步深入介绍这个模块的其他特色之前，还需要指出，`File::Find::Rule`也支持基于迭代器的接口。这对于筛选结果非常多的情况很有益处。比如要想找出我的笔记本中的所有文件，结果会是非常巨大的数组，应该有上百万的记录。把这样的数据存放在内存里面并不合适，另外采集这些数据也颇费时间。有时候在读取文件的同时执行操作更加方便，这样就不用等到遍历结束才开始处理。要使用这个迭代特性，我们只需要把方法列表开头（或者末尾，看你从左还是从右开始计算）的`in()`替换成`start()`：

```
my $ffr = File::Find::Rule->file()->name('*.pl')->start('.');
```

这段代码会返回一个对象，它带有一个`match()`方法。这个方法会在发现了某个匹配文件的时候返回（当然也可能返回`false`，那意味着搜索结束）：

```
while ( my $perl_file = $ffr->match ){
    # 做一些处理 $perl_file 文件的工作
}
```

这让你很容易逐个处理那些搜索结果（非常类似之前看到的`wanted()`子例程，只是更加简单，因为现在你只需要关心匹配的情况）。

现在介绍使用`File::Find::Rule`的第二个好处。你可能已经猜到了，那就是可以构建非常复杂的过滤方法链，用来精确定位需要处理的文件。如果你想要找到某几个用户所拥有的特定大小的、可执行的Perl文件，那是小事一桩。代码如下：

```
use File::Find::Rule;

@interesting =
    File::Find::Rule
        ->file()
        ->executable()
        ->size('<1M')
        ->uid( 6588, 6070 )
        ->name('*.pl')
        ->in('.');
```

如果你已经看过`File::Find::Rule`的说明文档，那么你可能已经注意到，虽然默认的链拼接方式是逻辑“与”（也就是说必须都匹配才行），`File::Find::Rule`其实还允许你用`or()`方法或者`any()`方法来完成“这样或那样”或者“最起码得怎么样”的搜索。另外，可能你还注意到`grep()`方法能够帮你打开文件、检查内容，而且它也是一个过滤方法。但这还不算是最酷的事情。

Richard Clamp已经把这个模块设计成可以允许其他用户任意加入自定义过滤方法。乍看这并不算吸引人，但看过那些“开罐可食的”、五花八门的过滤模块之后，情况就大不相同了。这里列出几个自定义过滤方法供参考：

- `File::Find::Rule::VCS`的作者是Adam Kennedy，他加入的方法可以轻易跳过各种源代码控制系统（比如CVS、Subversion和Bazar）生成的控制文件。
- `File::Find::Rule::PPI`也是同一作者的杰作，它让你能够搜索Perl文件中的特别Perl元素（比如搜索所有那些带有POD文档的文件，甚至搜索所有用了子例程的文件）。这并非简单地对文件执行`grep()`，而是真正地对文件进行语法解析。
- `File::Find::Rule::ImageSize`，是Richard Clamp所作的一个小插件，它能让你按照图片大小来挑选文件。
- `File::Find::Rule::Permissions`的作者是David Cantrell，它让你按照用户的权限来搜索文件（或目录）。比如检查某个目录下有没有`nobody`可以修改的文件。
- `File::Find::Rule::MP3Info`的作者是Kake Pugh，它让你按照任意MP3标签来搜索音乐文件。比如搜索某位歌手的所有音乐，或者搜索时长60分钟以上的乐曲。在最后一章我们会使用到这个模块。

这个家族里面还有很多模块没有介绍。它们往往比这里介绍过的模块更加通用（例如允许按照文件权限和日期来搜索），这里介绍的模块只是为了让你知道这个创意的潜力所在。

操纵磁盘限额

类似`core`删除程序这样的脚本很有用，能避免文件撑满磁盘。但是即使定时运行这个程序，仍然是一种被动的解决方案，系统管理员只能在文件形成（并填满磁盘）以后才能发现并删除它们。

其实还有一个更加主动的解决方案：磁盘限额。文件系统限额能够限制某个用户在某个文件系统上消耗的空间总数。书中提到的所有操作系统都能以某种形式实现磁盘限额。

主动方案这类手段往往显得比较强硬，因为磁盘限额会对所有文件一视同仁，并非只针对`core dump`这样的垃圾文件。大多数系统管理员发现，部署自动清理脚本并同时配置磁盘限额才是最佳策略，因为清理脚本的存在能提高磁盘限额的“柔性”。

在这一节，我们主要用Perl来操纵Unix限额（在这一章的末尾我们会快速介绍NTFS限额）。在编写Unix限额脚本之前，我们必须先介绍限额是如何被“手动”设定并查询的。Unix系统管理员一般是通过在文件系统挂载表（比如`/etc/fstab`或`/etc/vfstab`）中加入记录项并重新启动系统或手动调用`quotaon`这样的限额启用命令。下面就是Solaris系统中常见的`/etc/vfstab`文件：

#device	device	mount	FS	fsck	mount	mount
#to mount	to fsck	point	type	pass	at boot	options
/dev/dsk/c0t0d0s7	/dev/rdsk/c0d0t0d0s7	/home	ufs	2	yes	rq

最后一列的rq选项能在这个文件系统上启用磁盘限额。限额是按照每个用户来统计的，查看某个用户在所有文件系统上限额的命令是quota，调用方法如下：

```
$ quota -v sabrams
```

这会产生类似下面的输出：

```
Disk quotas for sabrams (uid 670):
Filesystem  usage  quota   limit  timeleft  files  quota  limit  timeleft
/home/users 228731 250000 253000          0      0      0
```

我们只对前三列输出感兴趣，它们在后面的例子中会被用到。第一列数据是用户sabrams在挂载于/home/users的文件系统上当前消耗的磁盘空间（以1 024字节为单位）。第二列是此用户的“软限额”。软限额达到以后，操作系统会不断提醒用户，但是并不会拒绝新的磁盘分配请求。第三列是“硬限额”，是用户磁盘用量的绝对上限。一旦这个限额达到，任何以此用户身份提出的分配空间请求都会被拒绝，错误信息应该类似于“超出磁盘限额”。

如果我们想要手动修改磁盘限额，一般会调用edquota命令。这个命令会带你进入某个编辑器（由EDITOR环境变量指定的编辑器），通过修改其中预先载入的临时文本文件来调整限额。下面的例子显示了某个用户在四个启用限额的文件系统的磁盘限额。用户很可能把home目录设在了/exprt/server2，因为他只能在那个文件系统上分配空间。

```
fs /expirt/server1 blocks (soft = 0, hard = 0) inodes (soft = 0, hard = 0)
fs /expirt/server2 blocks (soft = 250000, hard = 253000) inodes (soft = 0, hard = 0)
fs /expirt/server3 blocks (soft = 0, hard = 0) inodes (soft = 0, hard = 0)
fs /expirt/server4 blocks (soft = 0, hard = 0) inodes (soft = 0, hard = 0)
```

通过edquota来设置用户限额，这对于单个用户来说还算是个不错的方法，但是对十个、百个或者上千个用户显然不是最佳的方法。不过马上你会发现，还有更好的玩法。

Unix的弱点之一在于缺少修改限额的命令行工具。大多数Unix衍生系统都有C语言级别的库例程用来支持这个任务，但没有哪个Unix衍生系统带有常见的命令行工具来支持脚本化需要。我们深信Perl的座右铭“条条大路通罗马”（There’s more than one way to do it, TMTOWTDI，读起来类似“tim-toady”），因此我们会展示两种实现方法，使用edquota和使用Quota模块。

最新趋势？

实际上还有第三种方法，但是我不打算演示它，因为这个方法的可移植性很差。随着时间推移，有些Unix衍生系统已经开始提供（Perl可调用的）修改磁盘限额的命令行工具。例如很多系统都引入了`setquota`命令。然而这个趋势并没有被广泛接受，而且此命令往往有与平台相关的命令行参数，在使用时可能遇到其他问题。

另外还有一个名为`quotatool`的软件包，作者是Mike Glover，维护者是Johan Ekenberg。这个软件包计划实现更好的跨平台限额修改功能。可以在<http://quotatool.ekenberg.se>找到它。

`quotatool`已经非常有吸引力了，不过我还是打算演示如何操纵`edquota`，原因有二：首先`quotatool`可能没有移植到你所用的系统；其次（也是更重要的），`edquota`相关的技巧非常值得学习，它往往能在很多其他场合派上用场。

通过edquota技巧来编辑磁盘限额

手动设定用户的磁盘限额时发生了很多事情。我得先说明它的原理：首先，`edquota`命令会调用编辑器来编辑一段文本，并且用文件前后版本的差异来修改系统的限额记录项。其实，在这个过程中并没有规定修改必须通过编辑器的键盘输入来产生，甚至没有规定启动的必须是何种编辑器。`edquota`所需要的其实只是一个可以启动的程序，并且这个程序能够修改一小段文本文件。任何有效路径（由`EDITOR`环境变量指定）指向的程序都是可行的。为什么不在Perl脚本中指向`edquota`呢？下面的例子中，我们会展示这个特殊的脚本。

我们的脚本需要做两件事情。首先，它需要获取用户的命令行参数，正确设置`EDITOR`，然后调用`edquota`。于是`edquota`会再次调用我们的脚本，而这次我们会实际编辑临时文件。图2-1展示了这一系列事件的顺序。

最初的程序调用必须告诉第二个副本要修改什么。然而这个信息如何传递并不是非常直观。`edquota`的手册里面说：“如果没有设置`EDITOR`或者其他的环境变量，则会调用`vi(1)`。”确实可以通过`EDITOR`或者其他环境变量来传递命令行参数，但这样做并不可靠，因为我们不知道`edquota`会如何对待这样的环境变量。实际上，我们会通过Perl进程间通信机制来完成修改信息的传递。请参考补充内容“能谈谈么？”来了解更多可行的机制。

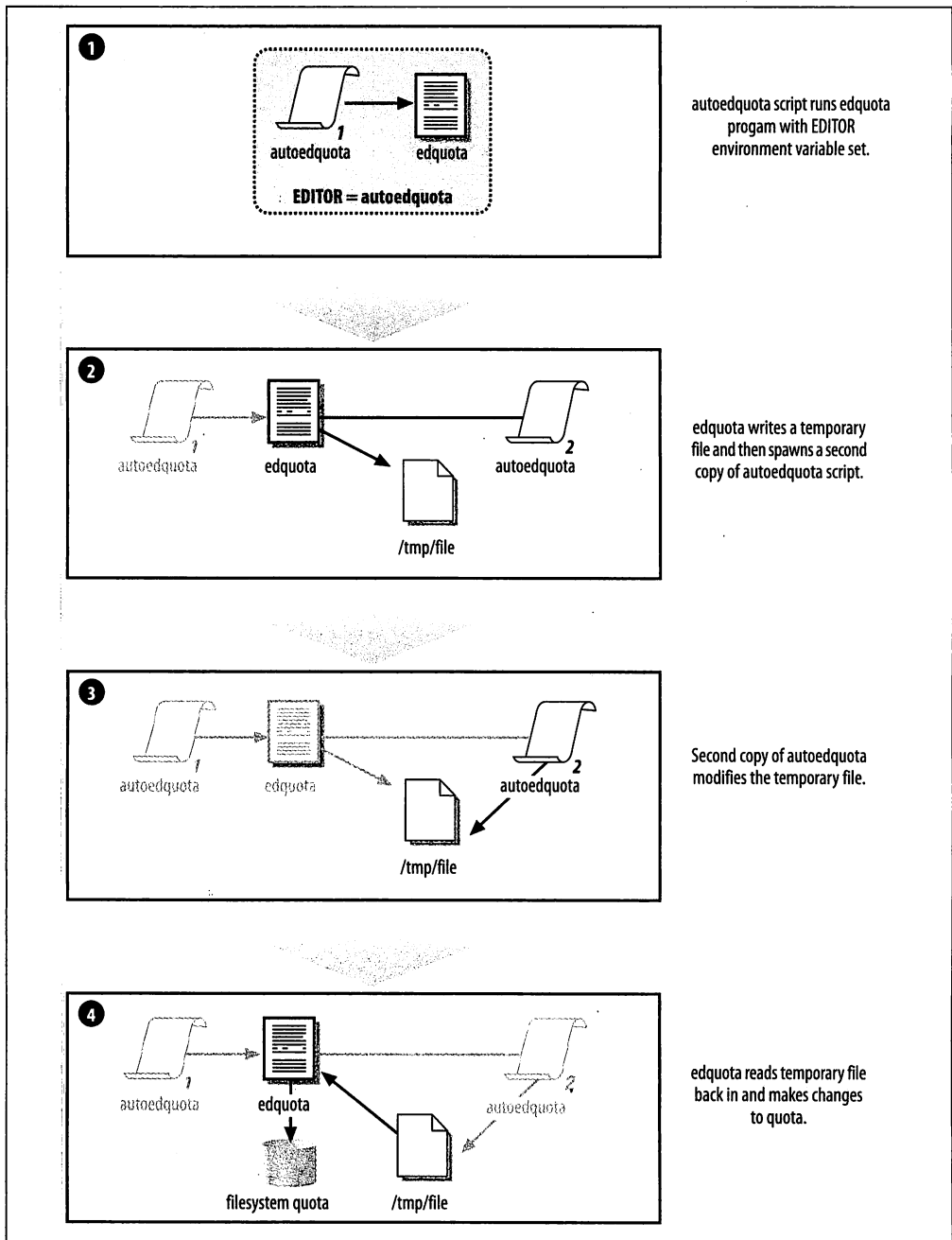


图2-1：通过“花招”修改限额

在我们的案例中，需要一种简单而有效的交换信息的机制。因为第一个进程只需要向第二个进程传送简单的几个修改指令（哪个限额需要修改以及限额的新值），我们会在两

者之间建立标准Unix管道。^[注5] 第一个进程会把修改指令写进管道，而edquota产生的副本会从读取管道内容作为标准输入。

能谈谈么？

在两个Perl进程需要互通信息的时候，可以选择以下几种形式，比如：

- 通过临时文件来传送。
- 使用命名管道来交换信息。
- 在Mac OS X中，可以使用AppleEvents。
- 在Windows中，可以使用互斥量（mutex）或者双方约定的注册表项。
- 使用网络socket传送。
- 使用共享内存区段。

作为程序员，你可以选择适合的通信协议。不过，有时数据本身已经决定了最适合的协议。

面对这些数据的时候，你应该考虑：

- 通信的方向（单向还是双向？）
- 通信的频率（单个信息的传送，还是有很多信息包要传送？）
- 数据的大小（是10MB的文件还是20个字符的短信息呢？）
- 数据的格式（是二进制文件还是纯文本呢？是定长的信息还是字符分隔的变长信息？）

最后，当然还要考虑脚本程序开发时的复杂度。

现在开始编写程序。脚本启动后的第一件事就是检查它要扮演的角色。第一个调用应该是用来读取命令行参数（获取需要修改的内容）；而第二个调用则由edquota启动，它感兴趣的应该是临时文件的文件名。这样我们可以假定：只要程序启动时带有多多个命令行参数，那么它就是第一个调用。这样，我们可以进一步验证它必需的启动参数。以下就是用来判定启动角色（例如，它是作为\$EDITOR被调用的）以及如果需要的话处理调用edquota的代码：

```
#!/usr/bin/perl
```

注5：实际上管道是通往edquota程序的，而edquota程序会帮我们把它的输入和输出定向到新的Perl脚本实例。

```

use Getopt::Std;
use File::Temp qw(tempfile);

my $edquota = '/usr/sbin/edquota'; # edquota 路径
my $autoedq = '/bin/editquota.pl'; # 此脚本的完全路径
my %opts;

# 这是否是此脚本的首个调用?

# 如果命令行参数操作不止一个, 那么就是首个调用
# 这样的话, 可以分析所有参数, 然后调用edquota程序

if ( @ARGV != 1 ) {

    # 冒号 (:) 表示该标志可以接受参数
    # $opts{u} = 用户 ID, $opts{f} = 文件系统名称
    # $opts{s} = 软限额大小, $opts{h} = 硬限额大小
    getopt( 'u:f:s:h:', \%opts );

    die "USAGE: $0 -u <uid> -f <filesystem> -s <softq> -h <hardq>\n"
        unless ( exists $opts{u}
            and exists $opts{f}
            and exists $opts{s}
            and exists $opts{h} );

    CallEdquota();
}

# 否则的话, 我们就是第二个调用, 那么应该执行限额编辑
else {
    EdQuota();
}

```

通过管道来调用edquota的代码非常简单:

```

sub CallEdquota {
    $ENV{'EDITOR'} = $autoedq; # 设置EDITOR变量指向脚本自己

    open my $EPROCESS, '|-', "$edquota $opts{u}"
        or die "Unable to start $edquota: $!\n";

    # 把修改内容送给第二个脚本调用
    print $EPROCESS "$opts{f}|$opts{s}|$opts{h}\n";

    close $EPROCESS;
}

```

以下是作为编辑器时的动作 (也就是调用edquota之后的代码):

```

sub EdQuota {
    my $tfile = $ARGV[0]; # 从参数中获得edquota的临时文件的名字

    open my $TEMPFILE, '<', $tfile
        or die "Unable to open temp file $tfile:$!\n";
}

```

```

my ( $SCRATCH_FH, $scratch_filename ) = tempfile()
    or die "Unable to open scratch file: $!\n";

# 从首个调用中获得修改信息行，并且移除末尾的换行符
chomp( my $change = <STDIN> );
my ( $fs, $soft, $hard ) = split( /\|/, $change ); # 分析出限额的值

# 读取临时文件的每一行，看看其中是否有需要修改的文件系统
# 有的话就修改它的限额并且存入草稿文件
while ( my $quoteline = <$TEMPFILE> ) {
    if ( $quoteline =~ /^fs \Q$fs\E\s+/ ) {
        $quoteline
            =~ s/(soft\s*=\s*)\d+(, hard\s*=\s*)\d+/$1$soft$2$hard/;
    }
    print $SCRATCH_FH $quoteline;
}
close $TEMPFILE;
close $SCRATCH_FH;

# 用草稿文件来替换临时文件
# 以便 edquota 获得修改后的结果
rename( $scratch_filename, $tfile )
    or die "Unable to rename $scratch_filename to $tfile: $!\n";
}

```

警告： 这个代码只能在以下条件下运行：

1. 脚本的开始行是以“shebang”（也就是#!）开头的，这行声明了文件是由Perl来解释执行的，不是常规的shell脚本。
 2. 文件本身设置了可执行属性：`chmod o+x /bin/ editquota.pl`
-

这里的代码是基本框架，但它仍提供了一种方法让限额的修改自动化。如果你之前一直是手动修改限额的话，那么这对你来说应该是个好消息。请在代码中加入错误检查和并发修改的保护机制，然后再把它部署到生产环境中。无论如何，相信你会在其他场合找到这种花招的应用机会。

使用 Quota 模块来编辑限额

曾经有段时间，前面这种方法是唯一能对限额修改进行脚本化的手段。然而 Perl 的 XS 扩展机制允许我们把 C 语言库程序中的限额支持集成到 Perl 里面，所以 Quota 模块的创建其实只是时间问题。我们要感谢 Tom Zoerner 和其他人的努力让它成为现实，现在设置限额变得易如反掌（只要你的 Unix 支持这个模块）。如果目前还不能支持，请使用之前的技巧。

下面是一段采用相同参数的限额编辑脚本：

```

use Getopt::Std;
use Quota;

```

```

my %opts;
getopt( 'u:f:s:h:', \%opts );
die "USAGE: $0 -u <uid> -f <filesystem> -s <softq> -h <hardq>\n"
    unless ( exists $opts{u}
        and exists $opts{f}
        and exists $opts{s}
        and exists $opts{h} );

my $dev = Quota::getqcarq( $opts{f} )
    or die "Unable to translate path $opts{f}: $!\n";

my ( $curblock, $soft, $hard, $btimeo, $curinode, $isoft, $ihard, $itimeo )
    = Quota::query( $dev, $opts{u} )
    or die "Unable to query quota for $opts{u}: $!\n";

Quota::setqlim( $dev, $opts{u}, $opts{s}, $opts{h}, $isoft, $ihard ) == undef
    or die "Unable to set quota: " . Quota::strerr() . "\n";

```

在分析参数之后，还有三个简单的步骤。首先是使用`Quota::getqcarq()`来获取设备的标识符，此标识符可以作为参数调用其他的例程。然后我们用设备编号和用户 ID 作为`Quota::query()`的参数来获取当前的限额设定，我们需要从其中挑出感兴趣的字段。最后设定限额。这样我们只用三行代码就完成了任务。

记住，Perl 的座右铭是“条条大路通罗马”，而不是“条条都是康庄大道（several equally good ways）”。

在Windows下编辑NTFS限额

说起基于Windows系统的磁盘限额实际上有两层含义。在最基础的层面，每个NTFS文件系统都可以按照用户和卷来控制限额（也就是说，某个特定用户只能在X卷上拥有Y大小的空间）。用户可以是本地用户也可以是Active Directory中可见的用户。Windows Server 2003 R2又进一步设置了卷和目录级别的限额，但是与特定用户无关。

第二层的限额是对一批机器上的用户起作用。这种情况下，按照每台机器的组策略对象（group policy object, GPO）来设置某个管理单元（organizational unit, OU）内的多台机器上的限额策略。

这一节里我们只关注第一层的限额，因为设置并维护GPO实在是有点离题太远。要想了解更多关于使用Perl来维护GPO的信息，我推荐你看看Robbie Allen和Laura Hunter合著的《Active Directory Cookbook》（O'Reilly出版）。另外为了限制篇幅，我们把对代码的解释精简到最少。这里使用了Windows管理规范（Windows Management Instrumentation, WMI），我们将在第4章深入介绍这种技术。如果你对WMI不熟悉，我建议你把这一页标注出来，等到仔细阅读了第4章关于WMI的介绍之后再回来。

下面的代码为一个WINDOWS域中名为`dnb`的用户创建限额记录。创建的记录中设置了本机C:卷的限额量（或者，如果已经存在这样的记录，限额量就会被改写）：

```
use Win32::OLE;

my $wobj = Win32::OLE->GetObject('winmgmts:\\\\.\root\cimv2');

# 下一行代码对Vista来说需要用提升权限执行
my $quota
    = $wobj->Get(
        'Win32_DiskQuota.QuotaVolume=\'Win32_LogicalDisk.DeviceID="c:"\'',
        'User=\'Win32_Account.Domain="WINDOWS",Name="dnb"\'');
$quota->{Limit} = 1024 * 1024 * 100; # 100MB
$quota->{WarningLimit} = 1024 * 1024 * 80; # 80MB
$quota->Put_;
```

这段脚本先获取引用WMI命名空间的对象。然后用这个对象来获取另一个代表用户磁盘限额的对象，也就是某某域里面某某用户在某个卷上的限额对象。然后用这个新的对象来设置两个属性（Limit和WarningLimit），并用Put_方法来使设定生效。^[注6]如果我们只是想知道限额的大小，可以简单打印后就返回，不必调用Put_方法。注意，想要让代码在Vista下运行，需要以提升权限（elevated privilege）来运行脚本（不是切换到管理员账户直接运行），请参考第1章获取更多信息。

查询文件系统使用量

我们刚刚介绍了限制文件系统使用的方法，现在自然需要检查它是否工作。让我们了解一下在本书所涉及的几个操作系统中如何查询文件系统的使用量。

想要在Windows机器上查询文件系统使用情况，可以使用Mike Blazer的Win32::DriveInfo模块：

```
use Win32::DriveInfo;

my ($sectors, $bytessect, $freeclust, $clustnum,
    $userfree, $total, $totalfree
) = Win32::DriveInfo::DriveSpace('c');

# 如果已经开启了磁盘限额，从用户角度出发，我们可以直接显示 $userfree
# 而不必显示“$total 字节中的 $totalfree 可用”
print "$totalfree bytes of $total bytes free\n";
```

Win32::DriveInfo 还能提供其他相关信息，比如哪个磁盘卷处于激活状态、某个卷（比如CD-ROM）是否在使用中，所以它很有用处。

注6：如果你有兴趣的话，设置磁盘无限制使用的方法是把限额设置为18446744073709551615，我没有开玩笑，微软的Scripting Guy就是这么说的。可以参考<http://www.microsoft.com/technet/scriptcenter/resources/qanda/jan08/hey0128.mspx>。

还有不少Unix模块可用，比如Filesys::DiskSpace（作者是Fabien Tassin）、Filesys::Df（作者是Ian Guthrie）和Filesys::DiskFree（作者是Alan R. Barclay）。前两个模块是通过Unix系统调用statvfs()来工作的，而第三个则是调用各种Unix平台的df命令并分析输出信息。挑选模块更多是看个人喜好以及平台支持情况。我个人喜欢Filesys::Df，因为它功能丰富，并且不会启动另外一个进程（这可能导致安全问题，参考第1章）来完成查询。以下代码能完成和前面的范例程序相似的功能：

```
use Filesys::Df;

my $fobj = df('/');

print $fobj->{su_bavail}* 1024 . ' bytes of ' .
      $fobj->{su_blocks}* 1024 . " bytes free\n";
```

我们需要做些数学计算，也就是*1024。因为Filesys::Df返回的数值是块的数量，我们的系统上块的大小是1 024个字节。如果需要，这个模块的df()函数也可以带一个可选的块大小参数。值得注意的是这里的两个哈希值，su_bavail和su_blocks，它们分别对应可用磁盘空间和磁盘总空间。这两个值都没有像Unix的df命令那样做10%的管理员“克扣”。如果我们想要看一般用户能见到的信息，可以使用user_blocks和user_bavail。

Guthrie还写了一个相关的模块，名为Filesys::DfPortable，这个模块有类似于Filesys::Df的语法，并且加入了对Windows的支持。如果你不需要Win32::DriveInfo提供的那些额外功能，这个模块可能会让你满意。

有了这里介绍的Perl代码，我们很容易构造出功能强大的磁盘监控和管理系统。相信这些脚本能避免很多磁盘问题的发生。

本章所用模块

模块名	CPAN ID	版本
MacOSX::File	DANKOGA	0.71
File::Find（随Perl发布）		1.12
File::Spec（随Perl发布，作为PathTools模块的一部分）	KWILLIAMS	3.2701
Path::Class	KWILLIAMS	0.16
Cwd（随Perl发布，作为PathTools模块的一部分）	KWILLIAMS	3.2701
Win32::File（随ActiveState Perl发布）	JDB	0.06
Win32::FileSecurity（随ActiveState Perl发布）	JDB	1.06
File::Basename（随Perl发布）		2.76
File::Find::Rule	RCLAMP	0.30

模块名	CPAN ID	版本
Getopt::Std (随Perl发布)		
File::Temp (随Perl发布)	TJENNESS	0.20
Quota	TOMZO	1.6.2
Win32::OLE (随ActiveState Perl发布)	JDB	0.1709
Win32::DriveInfo	MBLAZ	0.06
Filesys::Df	IGUTHRIE	0.92
Filesys::DfPortable	IGUTHRIE	0.85

更多参考资料

对Perl程序员来说，关于不同平台上的差异到底有哪些的问题，最好的回答便是“*perlport*帮助手册”。

《Active Directory Cookbook》(Secord Edition)，由Robbie Allen和Laura Hunter合著，以及《Windows Server Cookbook》，由Robbie Allen编写的（这两本书都来自于O'Reilly），都提供了许多殷实的代码范例，教你如何在Windows操作系统上编写相应的脚本程序，包括文件系统方面的特殊处理和操作。Allen架设了一个网站，提供了这两本书的代码范例仓库，所有代码都已经由他及合著者授权。在该站点上，你可以找到各种语言实现的脚本范例（包括由Perl改写的VBScript代码），你可以分别购买图书和对应的代码仓库。代码范例绝对是宝贵财富，该网站也是有关这类跨平台编程方面最为有用的网站了。我强烈建议购买这些代码范例（当然也包括这两本书）来支持作者付出的努力。

用户账户

下面是一个突击小测验，如果不是从用户角度出发，而是从系统管理员的角度来看，你会选哪个：

1. 更多好玩的功能
2. 努力避免问题

一般来说后者是这个问题的最佳答案，除非系统管理员刚刚被各种问题弄昏了头。如同我在第1章介绍的那样，系统管理员的职责是让技术对于用户来说尽量保持可用。

那么为什么会有这么多问题呢？用户是问题的来源，他们给系统管理和网络管理带来了更多复杂的事物：不确定性和不可重复性。这一章先讨论不可重复性，下一章会更多讨论不确定性。

一般来说，每个用户都希望系统能认识他。他们不仅仅需要唯一的名字，还需要拥有自己的“东西”。他们往往认为：“我有自己的文件，我会把它们放在我自己的目录下面。我打印文件时用的是我自己的打印资源。我把它展示在我自己的网页上面。”现在的操作系统也确实能够为每位用户维护这些信息。

但是在系统或网络系统中谁来维护这些账户信息呢？谁负责创建、保护并销毁这些个人账户呢？我可以大胆假定：这就是你，本书的读者——还有你开发的那些小脚本，它们可以作为你的工具。这一章就是为了帮你承担这个责任而写的。

让我们从介绍用户信息存放的位置开始，逐步深入用户管理。我们先关注 Unix 和衍生系统的用户，然后再分析基于 Windows 的操作系统用户。在讨论过两类操作系统之后，我们再尝试构造一个基础的账户系统。

Unix用户身份

在正式进入这个话题之前，我们需要花点时间先介绍几个用于存放用户身份的持久定义的关键文件。我们这里所说的“持久定义”是指那些存在于用户的整个生命周期的用户属性，哪怕用户没有激活或没有登录系统。这里的用户身份往往被称为账户（account），你可以用账户登录系统并成为系统的一员。

用户信息首次被加入密码文件（或某种目录服务库）的时候，这个用户就算被创建了。在记录被删除的同时，用户也被删除了。现在我们可以仔细看看用户身份是如何存储的。

经典Unix密码文件

我们先从经典密码文件开始介绍，逐渐过渡到更加复杂的文件格式。我们说这是经典的文件格式，是因为现在使用的Unix密码文件都是从它衍生出来的，包括Solaris、AIX以及Linux。这个文件一般被命名为`/etc/passwd`，是一个按行记录的ASCII文本文件，每行都记录了系统中的一个账户（或者指向另一个目录服务的链接）。行内的多个字段是通过冒号分隔的。稍后我们会介绍各个字段的含义及获取这些字段的方法。

以下是`/etc/passwd`的样本行：

```
dnb:fMP.o1mno4jGA6:6700:520:David N. Blank-Edelman:/home/dnb:/bin/zsh
```

至少有两种方法可以通过Perl来读取这行信息：

- 我们可以“手动”存取这个信息，把它当成一个常规文本文件来读取（并分析）：

```
my $passwd = '/etc/passwd';
open my $PW, '<', $passwd or die "Can't open $passwd:$!\n";

my ( $name, $passwd, $uid, $gid, $gcos, $dir, $shell );
while ( chomp( $_ = <$PW> ) ) {
    ( $name, $passwd, $uid, $gid, $gcos, $dir, $shell ) = split(/:/);
    <此处是你的代码>;
}
close $PW;
```

- 也可以让系统帮忙，通过Perl调用Unix系统库函数来完成文件的读取和分析。例如，下面的代码同样可以获取相同的信息：

```
my ( $name, $passwd, $uid, $gid, $quota, $comment, $gcos, $dir, $shell,
    $expire );
while (
    (
        $name,    $passwd, $uid, $gid,    $quota,
        $comment, $gcos,   $dir, $shell, $expire
```

```

    )
    = getpwent()
  )
  {
    <此处是你的代码>;
  }
endpwent();

```

使用系统调用的好处主要是能自动查询操作系统中启用的名称服务，比如网络信息服务（Network Information Service, NIS）、轻量级目录访问协议（Lightweight Directory Access Protocol, LDAP）、Kerberos或者NIS+。之后还会深入介绍这些库函数（包括使用getpwent()的简单方法），现在我们先看看代码返回的那些字段。

名称

登录login name字段里存放的是唯一代表机器上账户的短名称。之前见过的Perl函数getpwent()在列表上下文中返回很多字段，但在标量上下文中只返回名称字段：

```
$name = getpwent();
```

用户ID

在Unix系统中用户ID (UID)往往比用户名更加重要。系统中的每个文件都是被UID拥有的，并非被用户名拥有。如果我们把/etc/passwd中UID 2397对应的用户名从danielr改成drinehart，那么danielr所有的文件都会立刻显示成被drinehart拥有。可见UID是操作系统内部更加稳定可靠的用户身份标志。Unix内核以及文件系统都是用UID来记录文件的归属（以及资源的分配），并不依赖于用户名。如果有多个用户使用同一个UID，那么它们对系统来说其实是同一个用户。可以说用户名实际上是让用户能接触内核的便利方式。

下面是用来找到下一个可用UID的脚本。这个脚本会找出已经在使用中的最高UID，并且产生下一个可用的数字：

```

my $passwd = '/etc/passwd';
open my $PW, '<', $passwd or die "Can't open $passwd:$!\n";
my @fields;
my $highestuid;
while ( chomp( $_ = <$PW> ) ) {
    @fields = split(/:/);
    $highestuid = ( $highestuid < $fields[2] ) ? $fields[2] : $highestuid;
}
close $PW;
print 'The next available UID is ' . ++$highestuid . "\n";

```

注意：这个例子太过简单了，因为系统中常常有预先创建的高UID（为nobody、nfsnobody保留的），而且UID也有上限。另外，某些机构也有自己的UID分配策略（某些类型的用户必须使用某个范围的UID）。在编写这类代码时要考虑到这些例外。

表3-1列出了其他与登录名和UID相关的Perl函数和变量。

表3-1：与登录名和UID相关的Perl变量和函数

函数/变量	用处
getpwnam(\$name)	在标量上下文中返回登录名对应的UID，在列表上下文中则会返回密码记录的所有字段
getpwuid(\$uid)	在标量上下文中返回UID对应的登录名，在列表上下文中返回密码记录的所有字段
\$>	存放了当前Perl程序的有效UID
\$<	存放了当前Perl程序的真实UID

主要组ID

在多用户系统中，用户常常需要与其他用户共享文件和其他资源。Unix提供了用户组机制来支持这种需求。用户账户可以加入多个组，但必须属于一个主要的组。在密码文件中，主要组ID(GID)字段列出了用户属于的主要组。

组名、GID和组成员都存放在/etc/group文件中。这个文件存放了用户所属的次要组列表。要让某个用户属于几个次要组，只要把用户列在每个组对应的行后面就行了（但请注意，有些操作系统只允许用户最多加入8个组）。下面列出/etc/group中的几行作为例子：

```
bin::2:root,bin,daemon
sys::3:root,bin,sys,adm
```

第一个字段是组名，第二个字段是密码（有些系统中，用户加入组需要先输入密码），第三个字段是组对应的GID，最后一个字段是组的用户列表。

组ID分配的方案（scheme）往往是因地制宜的，因为每个位置都有自己特殊的管理划分和项目界限。常见的情况是以身份来分组（学生组、销售组），以角色分组（备份操作员组、网络管理员组），或者以账户分组（备份账户、批处理账户）。

Perl用来处理组文件的方式和处理密码文件的方式非常相似。我们可以把组文件当成一个标准文本文件来处理，也可以用特殊的Perl函数来完成。表3-2列出了组相关的Perl函数和变量。

表3-2：与组名和GID相关的变量和函数

函数/变量	用处
getgrent()	在标量上下文中返回组名，在列表上下文中返回\$name、\$passwd、\$gid和\$members字段
getgrnam(\$name)	在标量上下文中返回组ID，在列表上下文中返回和getgrent() ^a 相同的字段
getgrgid(\$gid)	在标量上下文中返回组名，在列表上下文中返回与getgrent()相同的字段
\$)	含有当前运行中Perl程序的有效GID
\$(含有当前运行中Perl程序的实际GID

a: 如果一个组的用户成员太多，以至于/etc/group的一行不够存放，列表上下文中getgrgid()和getgrnam()只会返回第一行的信息。这种情况下，需要通过不断调用getgrent()来构造成员列表。

“加密的”密码

到目前为止，我们已经介绍了与用户账户有关的三个关键字段，以及它们在Unix中的存放位置。现在要介绍的字段并非用户身份的标志，而是用来验证用户的身份、责任和权限。这是一个关于计算机如何知道某个用户是否是他自己的问题（是否可以把UID关联到某个用户名，比如说mguerre）。现在也有其他的身份验证手段（比如说公钥加密），但是密码字段这个机制自从Unix诞生开始就一直使用到现在。

经常看到密码文件中某一行的密码字段中只有一个星号(*)。因为标准的加密算法不可能产生只有星号的密码，所以通过编辑器设置这样的密码字段往往是用来锁住账户的。管理员通常会用这一招来拒绝用户登录，同时又不必删除账户。有时候会在密码串中加入一个星号，这样既可以锁住账户，又可以随时解开锁定（同时不必知道密码）。

注意：有时候还会用*LK*来表明账户被锁住，而*NP*或者NP用来表示密码不存在（尽管密码可能在其他地方，比如/etc/shadow，我们马上就会介绍）。

处理用户密码是一个单独的话题，请参考本书的第11章。

GCOS/GECOS字段

从计算机的角度来看GCOS/GECOS^[注1] 字段，它应该是最不重要的字段了。因为

注1： 想知道关于这个字段名字的有趣来历么？请参考 Jagdon Dictionary (<http://www.jargon.org>) 里面关于 GCOS 的介绍。

它往往用来存放用户的全名（比如“Roy G. Biv”），有时候还会把用户的头衔和电话分机号都存进去。

关心用户隐私的系统管理员应该特别注意这个字段。因为它是一个关联用户账户名和实名的数据源，而且很多时候它对于Unix系统的所有用户都是可读的，无论是在`/etc/passwd`文件还是目录服务库中。很多Unix程序（比如邮件客户端）会在添加涉及用户账户的信息时查询这个字段。如果你需要把某个用户从公众视野中藏匿起来（比如那个用户是持不同政见者、联邦证人或者著名人士），那么这是你需要经常关注的地方。

另外，如果你管理的那些用户并不是非常成熟，请考虑禁止用户随意设置GCOS字段值（同样要避免用户自己选择用户名）。相信你不希望在密码文件中看到咒骂的话或者其他任何不专业的信息。

Home目录

下一个字段是用户的`home`目录所在位置。这是用户登录并开始使用系统的时候会处在的目录。同时也是那些配置用户环境的文件存放的位置。

出于安全考虑，账户的`home`目录应属自己所有，并保持仅自己可写入，这非常重要。对所有账户开放可写入权限的`home`目录使得账户之间的越权攻击成为可能。有些时候甚至连用户自己可写入的`home`目录也能造成问题，比如对于那些受限制的账户（登录后只能完成指定任务，不能修改系统任何设定的账户）来说，可写入的`home`目录也可能导致越权问题。

以下代码能帮助确认每个用户的`home`目录只被用户自己所拥有且不能被他人写入：

```
use User::pwent;
use File::stat;

# 注意：这段代码对那些启用了自动挂载的home目录的机器来说
# 可能是非常重的负担
while ( my $pwent = getpwent() ) {

    # 确保我们操作的目录是真正的目录，而不是符号链接
    my $dirinfo = stat( $pwent->dir . '/' );
    unless ( defined $dirinfo ) {
        warn 'Unable to stat ' . $pwent->dir . " : $!\n";
        next;
    }
    warn $pwent->name
        . "'s homedir is not owned by the correct uid ('
        . $dirinfo->uid
        . ' instead '
        . $pwent->uid . " )!\n"
        if ( $dirinfo->uid != $pwent->uid );

    # 对于那些设置了 "sticky" 属性 (01000) 的目录来说，所有人可写入并不是问题
```



```

# 请参考 chmod 命令的手册来了解详细信息
warn $pwent->name . "'s homedir is world-writable!\n"
    if ( $dirinfo->mode & 022 and ( !$dirinfo->mode & 01000 ) );
}
endpwent();

```

这段代码和之前的文本文件分析的版本不太相同，这是因为它使用了Tom Christiansen的两个特殊模块：`User::pwent`和`File::stat`。这两个模块会重载经典的`getpwent()`和`stat()`函数，导致返回值产生戏剧性的改变。在载入了`User::pwent`和`File::stat`以后，这些函数开始返回对象而不再是列表。每个对象都有了一组方法，用来在列表上下文中返回某个字段的值。于是，查询文件元数据以获取其组ID的代码：

```
$gid = (stat('filename'))[5];
```

可以改成这样优雅的风格：

```

use File::stat;
my $stat = stat('filename');
my $gid = $stat->gid;

```

或者更加精简的版本：

```

use File::stat;
my $gid = stat('filename')->gid;

```

用户shell

传统密码文件中的最后一个字段是用户shell字段。这个字段通常会填入标准的交互式程序（*sh*、*csh*、*tcsh*、*ksh*或者*zsh*）中的一个。但其实也可以设置成任何可执行程序（或脚本）的全路径名。为了防止用户从守护进程（daemon）或锁住的账户登录，这个字段常常被设置成非交互式程序，比如`/bin/false`或者`/sbin/nologin`。

常常有人半开玩笑地说要把自己的shell设置成Perl解释器。有些人则已经开始认真考虑如何把Perl解释器嵌入*zsh*，当然这还得有些日子才能实现。然而，一些认真的人已经开发了类似<http://www.focusresearch.com/gregor/sw/psh/>和<http://www.pardus.nl/projects/zoidberg/>这样的Perl shell，并且把Perl嵌入了Emacs（参考<http://john-edwin-tobey.org/perlmacs/>），而Emacs号称编辑器中的操作系统。Perl也已经被嵌入了大多数新的vi编辑器衍生品中，比如*nvi*、*vile*和*Vim*。

也许你有理由开发一个非标准的交互式程序，并放在这个字段中供用户登录。比如说需要一个完全基于菜单操作的账户，那么可以把菜单程序填在用户shell字段中。但无论如何应该避免用户使用这个账户来获取真实shell并最终破坏系统。常见的错误是给用户一个邮件程序作为登录shell，并允许用户调入编辑器（或者pager）来读写邮件，而这个编辑器（或者pager）却带有内置的shell调用功能。

警告：要特别注意非标准的交互式登录程序。比如在允许用户使用ssh登录的系统中想要通过登录shell来锁住用户，那么你要特别注意SSH服务器不能配置成受用户的*.ssh/environment* 文件影响（对OpenSSH来说默认是关闭的）。如果启用这个文件，用户可以通过LD_PRELOAD弄出五花八门的漏洞。

可选的标准shell通常列在*/etc/shells*文件中。如果在*/etc/passwd*文件（或者网络版本的密码文件）中，用户的shell没有被列在*/etc/shells*文件中，大多数FTP守护进程会拒绝这样的用户登录。有些系统中的*chsh*会在接到用户更改shell的请求时查询这个文件作为验证。

以下的Perl代码能列出那些没有使用“许可shell”的账户：

```
use User::pwent;

my $shells = '/etc/shells';
open my $SHELLS, '<', $shells or die "Unable to open $shells: $!\n";

my %okshell;
while (<$SHELLS>) {
    chomp;
    $okshell{$_}++;
}
close $SHELLS;

while ( my $pwent = getpwent() ) {
    warn $pwent->name . ' has a bad shell (' . $pwent->shell . ")!\n"
        unless ( exists $okshell{ $pwent->shell } );
}
endpwent();
```

BSD 4.4对密码文件的改动

在BSD（Berkeley Software Distribution）4.3升级至4.4版本的过程中，传统密码文件有两处变动：在GID和GCOS字段之间引入了其他字段，另外还引入了二进制数据库格式来存储账户信息。

密码文件的额外字段

BSD 4.4版本引入的第一个字段是*class*，这个字段允许系统管理员对账户进行级别区分（可以对各种登录类（login class）设置不同的资源控制级别，比如CPU时间限制）。BSD分支还加入了*change*和*expire*字段用来标志密码修改期限和账户过期时间。在讨论下一个Unix密码文件格式时我们会介绍更多字段。

Perl也能支持这些密码文件中新加入的字段。有些操作系统提供了访问这些用户信息（比如磁盘限额和备注信息）的方法。只要在新的操作系统下编译了支持这些额外字

段，Perl就可以通过`getpwent()`来展现这些新字段。这也是一个推荐使用`getpwent()`而不建议直接对密码文件使用`split()`的理由。

二进制数据库格式

BSD 4.4还加入了一种数据库格式的密码文件，代替纯文本文件来存放账户信息。BSD的数据库格式比Unix的DBM (Database Management) 格式先进了很多。这样的改进提高了获取用户信息的速度。

`pwd_mkdb`程序用来读取文本格式的密码文件，然后将它转化成两个数据库文件，并把文本文件转移到`/etc/master.passwd`。这两个数据库文件提供了密码隐藏方案、不同的读取权限和密码的加密保护。下一节我们将进一步介绍这些内容。

Perl有能力直接读写这些DB文件（在第7章中我们会介绍如何实现），但我总建议不要在运行的系统中这样做。主要的问题在于写保护锁机制：要十分谨慎地读写这种关键数据库，并且确保当时没有其他程序在读写这个文件。标准的操作系统自带的程序（如`chpasswd`和`vipw`）会替你自动加锁。^[注2]另外，之前介绍过的（通过EDITOR变量“魔法”）修改磁盘限额的程序也能自动加锁。

影子密码

之前介绍了如何避免滥用GCOS字段信息，因为这个字段是所有用户可读的。另外一个非常敏感的公共字段是系统上所有用户的加密后的密码字段列表，尽管这些字段的信息已经通过加密保护起来了，但把它们暴露在所有用户的眼皮底下显然是有风险的，尤其是考虑到那些层出不穷的强力解密工具。^[注3]有些密码文件的字段是应该让所有人可读的（比如UID和登录之间的关系），而有些则不必。密码字段就是属于需要保护起来的字段。

常见的保护方法是把加密后的密码存储在一个特殊文件中，这个文件只有`root`用户才能读写，我们往往把这个文件称为“影子密码 (shadow password)”文件。在目前的操作系统发行版本中，这种做法已经成了标准。

在应用了影子密码文件之后，原始的密码文件需要引入一个小小的变动：密码字段被设置成一个特殊的字符来表明相应账户的影子密码已经生效。通常会使用的字符是`x`，而BSD数据库的`insecure`发行版使用的则是星号`(*)`。

注2: `pwd_mkdb` 在运行时可能会自动添加文件保护锁，也可能不会。这要看 BSD 的分支和版本而定，所以操作时请格外小心。

注3: 这里不会介绍高效的彩虹表 (rainbow table) 解密技术，有兴趣的话自己去看看吧 (http://en.wikipedia.org/wiki/Rainbow_table)。

注意：我听说有些影子文件会在这个字段中放入欺骗性的字符串。这对于那些尝试解密的人来说是个灾难，因为这个字符串和真实的密码没有任何关系。

大多数操作系统使用影子文件存放更多与账户有关的信息，类似于我们介绍过的BSD文件中的那些新字段（如账户有效期和密码有效期）。

在大多数情况下，Perl的密码函数（如`getpwent()`）能正确识别影子文件。因为只要C语言库程序能正常工作，Perl就可以。这意味着，如果你的Perl脚本在适当权限（也就是`root`）下运行，加密后的密码就能被正确获取，否则函数将不能正常工作。

不幸的是，Perl无法获取影子文件中其他字段的信息。Eric Estabrooks的`Passwd::Solaris`和`Passwd::Linux`模块可以读取这些信息（但受限平台）。如果这些字段对你来说很重要，或者你不希望通过他人的代码来读取敏感信息，那么我只能建议你忘记之前关于`getpwent()`的建议，而是手动读取影子文件并自己分析内容。

基于Windows的操作系统用户身份

现在我们已经展示了Unix系统中用来构成用户身份的信息源，是时候看看Windows的用户信息了。大多数信息在概念上是相似的，所以我们把注意力集中在两个操作系统的差异上。

在我们开始之前还有一件需要注意的事情：Windows系统在默认情况下会把用户信息存放在两个地方：本地存储（在本机存放，不与其他机器共享）或者域范围存储（很可能存放在域控制器的Active Directory库中）。在使用了域的情况下，这些信息会保存在用户的机器上，至少在用户的会话期间保存在那里。

和Unix用户信息的介绍类似，我们把注意力集中在本地账户信息上。想要了解如何与Active Directory或者其他的目录服务打交道，可以参考第9章。

Windows用户信息存储和访问

Windows把用户信息保存在一个名叫SAM（Security Accounts Manager，安全账户管理器）的数据库中，或者称目录数据库。SAM数据库是Windows注册表的一部分，位于`%SYSTEMROOT%\system32\config`。注册表文件存储时总是用二进制格式，这意味着常用的Perl文本分析功能无法奏效。理论上可以使用Perl的二进制处理函数（如`pack()`和`unpack()`）来处理SAM数据库，但是这需要在Windows不修改注册表的时候才能运行，而且这样做也有点太过疯狂。

幸运的是，Perl还有更好的方法来处理这些信息。

我们的解决方案是调用外部的二进制程序来帮助我们与操作系统交互。每台Windows的机器上都有一个功能强大的命令叫做net，你可以用它来加入、删除或者查看用户信息。net命令有些奇怪的地方，也比较难用，不过它还可以当作最后考虑的备选解决方案。

下面是net命令在某台有两个账户的机器上运行后的结果：

```
C:\> net users

User accounts for \\HOTDIGGITYDOG
-----
Administrator          Guest
The command completed successfully.
```

这个命令执行的结果很容易用Perl来分析。其他商业化的软件包提供的命令行可执行程序也可以完成同样的任务。

坏消息

这里有个坏消息，自从本书的第一版发行以来有些信息已经过时了。在第一版中我曾经推荐大家使用第三方模块来执行Windows上的用户管理任务，如Win32::UserAdmin（在O'Reilly出版的《Windows NT User Administration》一书中有所介绍），另外还有David Roth的Win32::AdminMisc和Win32::Perms（来自<http://www.roth.net/perl/packages/>），以及Jens Helberg的Win32::Lanman（藏在位于<http://www.cpan.org/modules/by-authors/id/J/JH/JHELBORG/>的他的CPAN目录中）。

据我所知，Win32::UserAdmin已经很久没人维护了。David Roth在1999年回到微软以后就没有再写Perl了。2005年我和David面谈时，他说非常乐于继续之前的工作，但是却找不到空闲时间来维护和改进这些模块的代码。他希望有其他人来帮忙维护代码，但直到目前为止没有发现志愿者。同样，Jens Helberg自从2003年之后就没有在Perl业界活跃过。

这些模块“年久失修”的现状让人担忧，因为它们曾经是能找到的最好的可用Windows模块。现在我不能继续推荐使用Win32::Lanman、Win32::AdminMisc或者Win32::Perms，因为它们的维护都不到位，但是如果你确实需要得到一份可用ActiveState发行版中的ppm加载的拷贝的话，可以考虑使用Win32::AdminMisc的5.10版本，可以从<http://www.ramtek.us>取得（Roth的站点也有Perl 5.8版本的Win32::AdminMisc和Win32::Perms），而5.8版本的Win32::Lanman可以从<http://www.bribes.org/perl/ppmdir.html>下载。

这本书会把主要兴趣集中在类似Win32API::Net这样的模块上。这些模块是官方libwin32模块集的组成部分，主要维护者是Jan Dubois和其他几个模块的作者。

另外，还可以考虑使用集成在ActiveState Perl发行版里面的Win32API::Net模块。以下是用来显示本地主机上的用户信息的代码。输出结果类似于Unix的/etc/passwd文件：

```
use Win32API::Net qw(:User);

UserEnum( '', \my @users );
foreach my $user (@users) {
    # 下面调用中的参数“3”意味着“User info level”，
    # 主要是指我们希望获得多少信息。
    # 这里我们要求返回第三级用户的详细信息。
    UserGetInfo( '', $user, 3, \my %userinfo );
    print join( ':',
        $user, '*', $userinfo{userId},
        $userinfo{primaryGroupId},
        '',
        $userinfo{comment},
        $userinfo{fullName},
        $userinfo{homeDir},
        '' ), "\n";
}
```

最后，你还可以使用Win32::OLE模块来访问内置于Windows中的活动目录服务接口（Active Directory Service Interfaces, ADSI）功能。我们会在第9章中进一步讨论相关的细节，所以这里先不举例子了。

稍后会展示更多用来访问并修改Windows用户信息的Perl代码，现在我们先比较Unix和Windows在用户管理上的差异。

Windows用户ID编号

Windows对待用户ID不是很严肃，但不允许用户ID被重用。在创建用户ID的时候，Windows仿佛是随机地找出一个唯一的用户编号（Windows中称为RID，也就是*relative ID*）。这个号码会与机器ID和域ID组合产生一串大数字，称为*security identifier*（安全标识符），或者简称为SID，它就是用户的用户标识符（UID）。如RID是500这样的数字，而由它作为一部分的SID看起来像这样：

```
S-1-5-21-2046255566-1111630368-2110791508-500
```

要知道，RID的值就在我们上面的代码中UserGetInfo()调用后的返回结果中。下面的代码可以用来打印某个用户的RID：

```
use Win32API::Net qw(:User);
```

```
UserGetInfo( '', $user, 3, \my %userinfo );  
print $userinfo{userId}, "\n";
```

删除某个用户之后，你不能使用任何常规方式来重新创建这个用户。哪怕你尝试使用同样的名字来创建，新用户也会拥有不同的SID，而他仍然不能访问之前那个用户的文件和其他相关资源。

这就是为什么Windows的书籍通常会建议为那些将要被其他用户继承的账户进行重命名。也就是说，如果新员工将要接收老员工的所有文件和权限，那么一般并不会创建新账户、转移文件并删除老账户，而是简单地进行账户重命名。我个人觉得这种做法有点不太妥当，因为这样新员工还会继承前任所有损坏的（或无用的）注册表信息。然而这个做法确实简单高效，而效率有时候真的非常重要。

这个建议被采纳的部分原因在于转让文件的所有权非常困难。在Unix中，高权限的用户能够这样下达命令：“把所有这些文件所有权都从这个用户转给那个用户。”但是在Windows中将所有权给予别人是不容易的，比较容易的是主动获得（也就是说作为管理员夺取文件所有权）。好在我们有两种方法来绕过这个限制并假装我们在使用Unix的语义。在Perl里面，我们可以：

- 调用一个特殊的二进制程序，如：
 - Cygwin发行包（在<http://www.cygwin.com>免费下载）中包含的chown二进制程序。如果你有Unix背景并使用Windows机器工作，你一定要看看Cygwin。要想使用商业版本的软件，也可以使用MKS Toolkit中包含的chown程序。
 - SubInACL二进制程序，可以从微软下载中心（<http://www.microsoft.com/downloads>）下载。这个软件的优势是来自微软，但是有小小的学习曲线。
 - 来自<http://setacl.sourceforge.net>的SetACL，非常像SubInACL，但又有自己的创新。如果你正考虑使用SubInACL，那么请先看看这个程序，因为它可能会更加友善一些。
- 使用Perl模块，例如：
 - Win32::Security，作者是TobyOvod-Everett。使用这个模块来修改某个文件的拥有者的代码如下：

```
use Win32::Security::NamedObject;  
my $nobj = Win32::Security::NamedObject->new('FILE', $filename);  
$nobj->ownerTrustee($NewAccountName);
```

Win32::Security有两个小秘密。首先，这个模块带有很多好用的工具脚本，比如说*PermDump.pl*可以用来显示继承的和直接获得的权限，而*PermFix.pl*可以用来修复那些因为文件删除而“失传”的权限。其次，这个模块在处理那

些既有permit ACL又有deny ACL（如果两者有相同的trustee）的对象时会有问题，所以在使用了deny ACL的情况下请谨慎使用。

- Win32::OLE，作者是Jan Dubois，用来调用WMI函数（请参考第4章中关于WMI的深入介绍）。这个方法有些难度，因为获取所有权很容易（也就是把所有者改为运行脚本的用户），而给出去却困难重重。^[注4] 获取所有权是通过TakeOwnership()来实现的，它是cimv2名称空间（namespace）中CIM_DataFile对象的方法。
- Win32::Perms的作者是Dave Roth，位于<http://www.roth.net/perl/packages>，文档在<http://www.roth.net/perl/perms>。（但请务必看看补充内容“坏消息”再决定是否要使用它。）下面的代码用这个模块修改了某个目录及其下文件（包括子目录）的所有者：

```
use Win32::Perms;

$my acl = new Win32::Perms();
$acl->Owner($NewAccountName);
my $result = $acl->SetRecurse($dir);
$acl->Close();
```

Windows密码和Unix密码不兼容

Windows常用的密码加密算法和Unix的并不兼容。一旦对明文密码进行加密，就不能在这两个平台之间进行复制（以保持同步），而这却是在Unix家族的操作系统（Linux、Solaris、Irix等）中常用的同步账户的手段。这对那些必须同时维护Unix和Windows环境的系统管理员来说是非常痛苦的。有些管理员通过定制的商业化用户认证模块（或其他解决方案）来绕过这个问题。

作为一名Perl程序员，如果不考虑使用定制的用户认证机制的话，唯一的出路就是让用户提供明文密码了。这样我们才可以分别在两类系统上同步维护用户的密码。

Windows组

直到目前为止，我们都能对本地主机存储的用户账户和网络服务（如NIS）中存储的账户一视同仁。对于我们之前遇到的问题来说，用户账户信息存储的位置并不重要。但是如果想准确地介绍如何用Perl来处理Windows用户组信息，我们必须更深入介绍。

在Windows系统中，用户的账户信息可以存储在两个位置：在机器的SAM数据库中或者

注4：我在各种语言中都找不到任何使用WMI来授予权限的例子，只能找到获取权限的。但我不能说这是不可能的，只能说我已经尽力了。

在域控制器的活动目录（Active Directory，AD）中。这使得那些只能在本机登录的本地用户与可以随意登录域中指定机器的域用户得以区分开来。用户信息通常同时存储在这两个地方，这使得用户可以从域中任意Windows机器上登录并保持对自己桌面环境的访问（就像文件服务器那样使用桌面），同时也允许用户在网络资源不用于认证或文件共享的情况下登录本机。

Windows 有四种用户组模型。区分它们的诀窍在于解答两个问题：组可以在哪里使用（也就是它的范围）以及组里可以包含什么（组的成员）。下面的列表先从列出最简单的“管辖权”开始，然后逐渐深入介绍：

本地组

只能在本机使用，用来控制对本地资源的访问。可以包含本地账户、域账户和全局组。

域本地组

可以用来控制域内资源的访问权限。可以包含账户、全局组、任何域内的组或者其他（同域内的）域本地组。

全局组

常用来被任意域内的其他组引入（包含）。可以包含账户和同一个域内定义的其他全局组。

通用组

可以用来引入同一AD实例内跨域和跨森林（即多组目录树）的组。可以包含账户、全局组和（同一森林内定义的）通用组。

本地组是与机器相关的。增加或删除本地组的操作很少见，大多数情况下只是需要修改默认组的成员。而作为对比，其他类型的组的使用方法则大不相同。

问题的关键在于组内嵌套其他的组。比如你想控制对某个共享资源的访问（最典型的例子是共享打印机）。其实大可不必列出能使用打印机的所有用户，更简单的方式是指定某个本地组的所有成员可以使用打印机。

你可能以为把用户添加进域本地组就可以让每个用户都能使用打印机，但这个解决方案逐渐开始显得不方便，因为某些部门可能正进一步细分职责。如果每次设施规划部门的员工入职时都必须分别加入三个打印机组、一个绘图机组，外加几个其他组，那么你这个系统管理员肯定会不开心。因为这样不但使得手工授权变得更加麻烦，而且也非常容易出错。

把所有资源都分配给设施规划部门这个组也不是什么好主意，因为可能会有多个组共享资源的可能性。好比现在设施规划部分人员太多，已经没有足够的办公空间，所以有些

人必须到另外一层楼去办公。而那些搬过去的人可能会需要使用设施规划部分所在楼层的打印机。如果打印机的使用权限和那一层的每个部门都必须关联起来，那么很快又回到了最初的问题，只不过这次需要罗列的是部门组。

解决这个问题的正确方法是用域本地组来嵌套全局组（比如设施规划部门组）。这样的设定能够让全局组内的成员自动获得合适的打印权限。之前的两个部门共享打印资源的问题也很容易解决，只要把相关的全局组加入域本地组就可以了。想要知道哪些组可以访问某台打印机的话，只要列出域本地组的成员就可以了。图3-1展示了组如何嵌套。

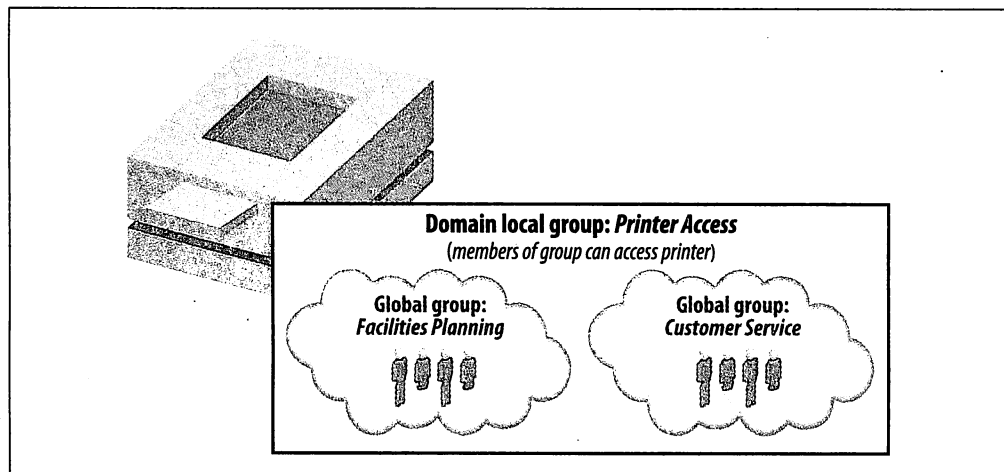


图3-1：Windows组如何嵌套

“全局”这个词有些不妥，因为听上去好像这样的组能够包含整个AD树里面所有的账户，但其实它只能包含本域的账户和本域内创建的组。而这正是通用组的长处，它让你能集成不同域内的全局组。如果你希望有一个统管各个域内所有账户的组，那可以尝试创建包含各个域内全局组的通用组。这样一来，你可以把这个通用组包含在某个“特权”组中，然后所有相关的用户都拥有了这样的权限。

可惜的是，这个解决方案对于Perl编程来说并不那么容易。主要是因为我们（也倾向于）使用各种不同的模块来处理不同类型的组。以下是可供选择的解决方案：

1. 如果要处理通用组，那么没有别的选择，只能使用Win32::OLE来进行ADSI调用。
2. 如果针对本地组、域本地组或者全局组工作，那么可以使用ADSI（通过Windows NT或者其他的LDAP供应商），或者可以使用之前介绍的模块Win32API::Net。通过Win32::OLE来调用ADSI的好处主要是提高了代码的一致性（统一使用一个模块），而使用Win32API::Net的好处则是代码很简洁（它带有专门的函数用于完成这个任务）。

让我们快速浏览各种实现方式。如果仍使用Win32API::Net，接下来要选择的无非是组的类型：本地组还是全局组？Win32API::Net针对不同组的处理函数也各不相同，它们全都列在表3-3中。

表3-3：Win32API::Net针对本地组和全局组的函数

本地函数	全局函数
LocalGroupAdd()	GroupAdd()
LocalGroupDel()	GroupDel()
LocalGroupAddMembers()	GroupAddUser()
LocalGroupDelMembers()	GroupDelUser()
LocalGroupGetMembers()	GroupGetUsers()
LocalGroupGetInfo()	GroupGetInfo()
LocalGroupSetInfo()	GroupSetInfo()
LocalGroupEnum()	GroupEnum()

第一列的函数能用来设置本地组（无论是针对机器的本地还是针对域的本地），而第二列的函数能设置全局组。这些函数的第一个参数用来制定修改范围。比如，要创建一个本机用户组，第一个参数就可以是''。要创建一个域本地组或者全局组，第一个参数应该是适当域控制器的名称。找出适当域控制器的方法是调用GetDCName()：

```
# $server 是需要查找其 DC 的服务器，
# $domainname 是你需要查找 DC 的域，
# $dcname 会保存找到的结果
GetDCName($server, $domainname, $dcname);
```

当然完成这个任务可能需要分别调用左右两列函数才行，看起来有点傻。例如要了解某个用户所属的所有组，需要分别为对本地组和全局组各调用一次函数，表3-3中的函数应该不用多解释。

下面是把某个用户加入全局组的例子：

```
use Win32API::Net qw(:Get :Group);

my $domain = 'my-domain';

# Win32::FormatMessage 将数字类型的错误代码转换为
# 我们能理解的东西
GetDCName('', $domain, my $dc)
or die Win32::FormatMessage( Win32::GetLastError() );

GroupAddUser($dc, 'Domain Admins', 'dnbe')
or die Win32::FormatMessage( Win32::GetLastError() );
```

注意：在Roth的书（参考本章末尾的那一节）里面有一个小窍门：访问本地组列表的代码必须在管理员权限下运行，而全局组的信息则是所有用户可见的。

如果需要用ADSI创建通用组，我们可以用下面的代码来完成（请参考第9章中的详细解释）：

```
use Win32::OLE;
$Win32::OLE::Warn = 3; # 要求错误信息更详细

# 参考了微软 ADSI 文档中的 ADS_GROUP_TYPE_ENUM
my %ADSCONSTANTS = (
    ADS_GROUP_TYPE_GLOBAL_GROUP      => 0x00000002,
    ADS_GROUP_TYPE_DOMAIN_LOCAL_GROUP => 0x00000004,
    ADS_GROUP_TYPE_LOCAL_GROUP       => 0x00000004,
    ADS_GROUP_TYPE_UNIVERSAL_GROUP    => 0x00000008,
    ADS_GROUP_TYPE_SECURITY_ENABLED   => 0x80000000
);

my $groupname = 'testgroup';
my $descript  = 'Test Group';

my $group_OU = 'ou=groups,dc=windows,dc=example,dc=edu';

my $objOU = Win32::OLE->GetObject( 'LDAP://' . $group_OU );
my $objGroup = $objOU->Create( 'group', "cn=$groupname" );
$objGroup->Put( 'samAccountName', $groupname );
$objGroup->Put( 'groupType',
    $ADSCONSTANTS{ADS_GROUP_TYPE_UNIVERSAL_GROUP}
    | $ADSCONSTANTS{ADS_GROUP_TYPE_SECURITY_ENABLED} );
$objGroup->Put( 'description', $descript );
$objGroup->SetInfo;
```

Windows 用户权力

Unix和Windows的最后一个用户身份的区别是“用户权力”的概念。在传统的Unix权力模式(schema)中，用户能够做的事情往往是被文件权限或者超级用户/普通用户身份所限制。而在Windows中，用户的权力可以在某种意义上“超越”自己的身份，并成为自己身份的一部分。^[注5] 比如说某个人可以把修改系统时间的权力给予某个普通用户，然后那个用户就能随意控制本机时间。

某些人可能会觉得用户权力的概念很令人困惑，因为他们往往用本地安全策略编辑器（Local Security Policy Editor）或者组策略/组策略对象编辑器（Group Policy/Group Policy Object Editor）来分析系统。“用户权力分配（User Rights Assignment）”（图

注5： 大多数现代 Unix 操作系统都可以使用 ACL 或者基于角色的访问控制（RBAC）来管理用户权力，但并不像 Windows 这么普遍。

3-2) 列出的策略往往和大多数人想看到的相反，它列出的是所有可能的用户权力，并期望着你把合适的用户和组加入其中。

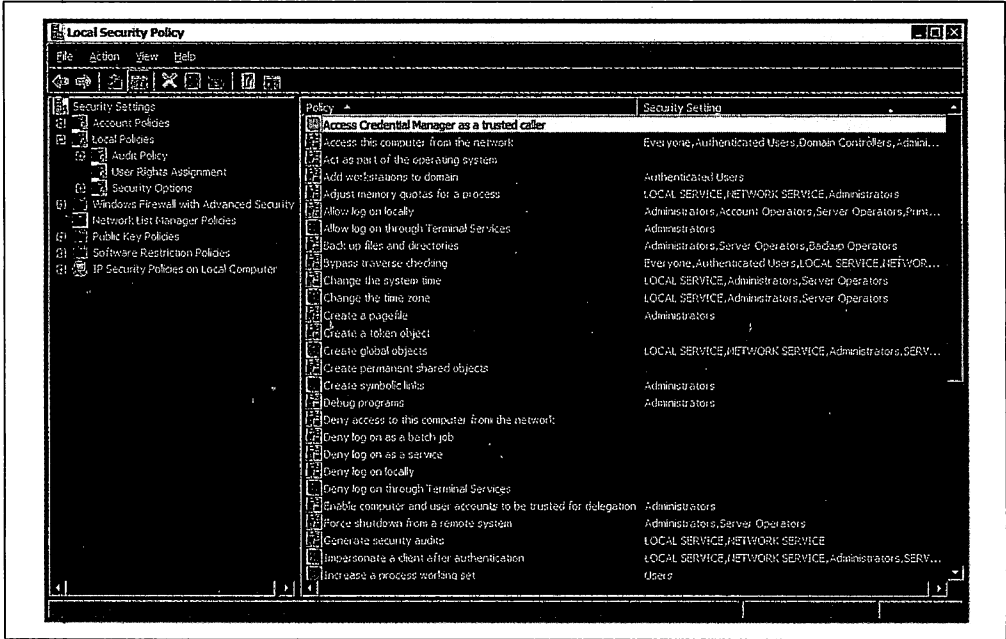


图3-2：通过本地安全策略编辑器来设置用户权力

更加合乎用户思路的用户界面应该是提供让用户添加权力或者删除权力的方式，而不是其他方式。^[注6] 这恰恰是我们使用Perl来分配权力的方式。

其中一个方式是调用来自Microsoft 2000/2003资源工具包（Resource Kit）里面的`ntrights.exe`程序。如果你还没有听说过资源工具包，请看看下面的介绍。

Microsoft Windows Resource Kits

“居家旅行必备Windows资源工具包”即使对严肃的Windows系统管理员来说也是值得青睐的，而且媒体也普遍认可它。微软出版社通常至少对每个OS版本出版一部大型工具书，其中包含了重要的参考信息。但其实让这部工具书值得期待的并不是信息，而是其中包含的那些重要的系统管理程序。这些工具让系统管理工作变得轻松很多。

注6：组策略管理器（在某种程度上）作为完成这个任务的新的用户界面，更加专注于接收权力的主体。它还提供了对 GPO 自动化操作的脚本编程能力。然而可惜的是，直到目前为止，还不可以直接对 GPO 进行脚本编程。

其中很多工具程序是操作系统开发人员编写的，他们编写这些程序的初衷是为了满足自己的需要。比如那些添加用户、修改文件系统安全信息、显示已安装打印机、处理漫游配置文件、帮助调试域和网络浏览服务等等的工具程序。

这些资源工具包里面的工具没有获得微软的支持，也就是说用户使用时必须完全自己负责。这虽然听上去有些可怕，但其实给了一个微软快速向管理员发布代码的途径。确实代码中会有些小bug，但实际并不影响使用。有些bug已经被修正，并已经发布在微软的网站上。

*ntrights.exe*使用起来非常简单，只要像使用任何其他程序那样（使用反引号或者system()函数）调用就可以了。这里只要这样调用*ntrights.exe*就可以了：

```
C:\> ntrights.exe +r <right name> +u <user or group name> [-m \\machinename]
```

这会给予某用户（或组）一个权力，可以带上机器名作为参数。反之要拿走某个权力：

```
C:\> ntrights.exe -r <right name> +u <user or group name> [-m \\machinename]
```

Unix用户可能会对这里使用+和-字符（如同在chmod中）的方式非常熟悉，在此处是前置于r开关。所有可授予的权力（如SetSystemtimePrivilege可以用于设定系统时间）名称列表可以参考资源工具包中ntrights命令的文档。如果你不想使用资源工具包，还可以考虑之前提到过的Cygwin软件包里面的editrights工具程序包。

另外一个方式是基于Perl模块的，这里用到的是Jens Helberg的Win32::Lanman模块。PPM包形式的可以在<http://www.bribes.org/perl/ppmdir.html>找到，而源程序形式的可以在位于<http://www.cpan.org/modules/by-authors/id/J/JH/JHELBERG/>的Helberg的CPAN目录下下载（这个模块在正规CPAN镜像中搜索不到，所以你必须直接到那个目录）。

注意：不论我之前怎么抱怨，这里还是得介绍Win32::Lanman这个模块，原因是我实在找不到任何替代品。无论我怎么找，实在没有办法找到类似WMI或者ADSI这样的办法来实现这个功能。我真的愿意证明自己错了！

我相信可以使用Win32::API来实现Win32::Lanman的功能，因为从Win32 API编程的角度来看它们是相通的，但以我的Windows编程水平来说这只是个猜想。如果哪个读者成功使用Win32::API重写了Win32::Lanman模块，请务必告诉我。我会在本书的下一版改为使用你的版本。

我们先用这个模块来查看某个用户的权力。这个任务有几个步骤，第一步是载入模块：

```
use Win32::Lanman;
my $server = 'servername';
```

下一步需要获得账户的SID，这里我们先获得*Guest*账户的SID：

```
Win32::Lanman::LsaLookupNames( $server, ['Guest'], \my @info )  
or die "Unable to lookup SID: " . Win32::Lanman::GetLastError() . "\n";
```

@info 现在包含一个引用了匿名哈希的数组：每个数组的一个元素都是一个用户账户信息的哈希。这个例子只有一个*Guest*账户。每个哈希以下的键：*domain*、*domainsid*、*relativeid*、*sid*和*use*。这里我们关注的是*sid*，现在我们把它取出来：

```
Win32::Lanman::LsaEnumerateAccountRights( $server, ${ $info[0] }{sid},  
    \my @rights )  
or die "Unable to query rights: " . Win32::Lanman::GetLastError() . "\n";
```

@rights数组现在包含了*Guest*用户拥有的所有权力。

要把用户权力和它们的API（Application Program Service，应用程序编程接口）名字对上号并非易事。最好的办法是参考软件开发工具包（Software Development kit，SDK）文档（位于<http://msdn.microsoft.com>）。这个文档很容易找到，因为Helberg通常是按照SDK函数名来命名自己的函数的。你可以在MSDN（Microsoft's Developer Network，微软的开发者网络）网站搜索“LsaEnumerateAccountRights”，应该很快能找到这个文档的链接。

给用户添加权力现在也很容易。比如说如果要允许*Guest*关闭主机，可以这么做：

```
use Win32::Lanman;  
my $server = 'servername';  
  
Win32::Lanman::LsaLookupNames( $server, ['Guest'], \my @info )  
or die "Unable to lookup SID: " . Win32::Lanman::GetLastError() . "\n";  
Win32::Lanman::LsaAddAccountRights( $server, ${ $info[0] }{sid},  
    [&SE_SHUTDOWN_NAME] )  
or die "Unable to change rights: " . Win32::Lanman::GetLastError() . "\n";
```

这里，我们在SDK文档里面找到了SE_SHUTDOWN_NAME，并调用由Win32::Lanman定义的&SE_SHUTDOWN_NAME子例程，这样就能得到 SDK 里面定义的常量。

Win32::Lanman::LsaRemoveAccountRights()函数用来删除用户权力，它的参数和前面添加权力的函数相似。

在开始下一个主题之前，有必要指出Win32::Lanman还另外提供了一个函数，而这个函数工作起来和本地安全策略编辑器难用的用户界面相似：它不是把权力绑定给用户，而是反过来把用户绑定到权力。不过有时候这个函数还是很有用的，比如可以通过Win32::Lanman::LsaEnumerateAccountsWithUserRight()来获取拥有某个特殊权力的SID列表。列举这个列表在某些情况下应该是有用的。

构建用户账户管理系统

现在我们已经有了关于用户账户的基础知识，是时候讨论管理用户账户的有关问题了。在这里，我不仅仅介绍用于管理账户的Perl子例程或函数，还希望在实际操作的层面上向你展示应用的“画卷”。在这一章的最后，我们会创建一个“基本框架（bave-bones）”风格的^[注7] 账户管理系统来同步管理Windows和Unix用户。

我们的账户系统有四个主要的部件：用户界面、数据存储、处理脚本（按照微软的说法是“商业逻辑”）和底层库例程。这四个部分相互配合完成任务（参考图3-3）。

添加账户的请求通过用户界面发布到系统并存放在“添加账户队列（add account queue）”文件中，简称为“添加队列”。一个处理脚本读取队列，完成用户账户的创建，并且把已创建的账户存放在另一个数据库中。这就完成了添加用户的流程。

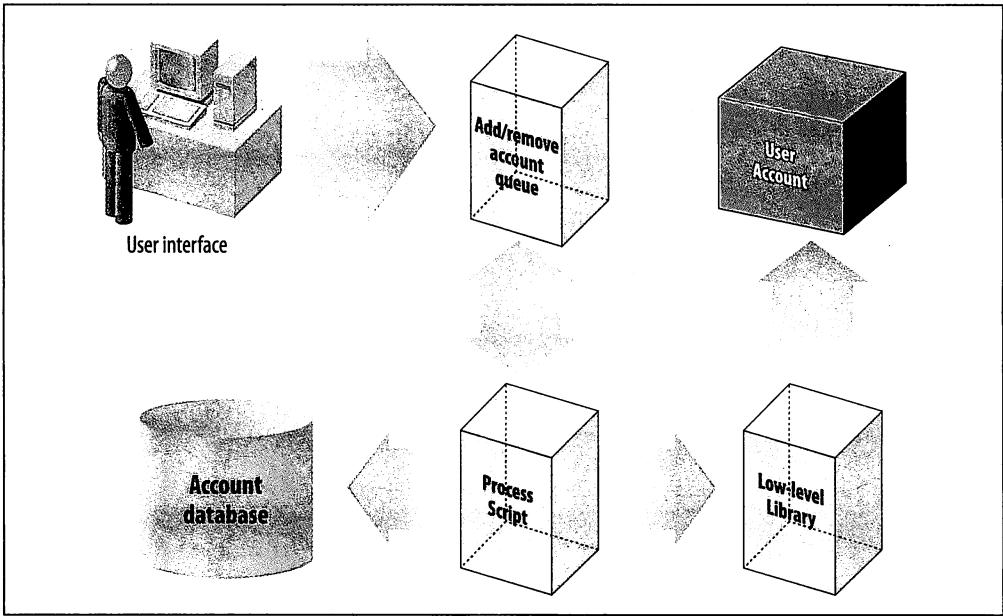


图3-3：账户管理系统的结构

删除用户账户的流程与添加流程相似。用户界面用于创建一个“删除队列”，然后由一个处理脚本读取队列，完成账户的删除，并且更新数据库的记录。

我们把问题分解成逻辑上互相独立的部分，这是为了最大程度地提高系统灵活性，以便

注7：“基本框架”也就是“快速原型”的同义词，这意味着我们关注的是通过简单编码来展示在系统构造背后的基础概念。

将来改进。假如某一天需要改变数据库后端，那么只需要修改相关的数据库例程。同样地，如果需要给添加账户的流程引入其他步骤（可能需要比对人力资源数据库），我们只需要修改处理脚本即可。

我们先查看第一个组件：用来创建初始账户队列的用户界面。因为这本书提倡“基本框架”编程，所以我们使用基于文本的用户界面来查询账户参数：

```
sub CollectInformation {
    use Term::Prompt;    # 我们稍后会拿掉这些 use 语句
    use Crypt::PasswdMD5;

    # 为演示需要而列出所有需要初始化的字段，
    # 这些字段理应通过配置文件来管理
    my @fields = qw{login fullname id type password};
    my %record;

    foreach my $field (@fields) {

        # 如果遇到密码字段，使用随机的 salt 值对它进行加密，然后存储
        if ( $field eq 'password' ) {

            # 请相信我，稍后我们会对存储哈希版本密码的这个决定后悔的，
            # 本章稍后会揭晓后悔的原因
            $record{$field} = unix_md5_crypt(
                prompt( 'p', 'Please enter password:', '', '' ), undef );
        }
        else {
            $record{$field} = prompt( 'x', "Please enter $field:", '', '' );
        }
    }
    print "\n";
    $record{status} = 'to_be_created';
    $record{modified} = time();
    return \%record;
}
```

这个例程创建了一个列表，并且用它来生成一个账户记录。如同注释提示的那样，这样硬编码的列表是为了简化代码。从软件设计角度来说，应该用配置文件来决定需要读入的字段名。其实，描述每个字段允许的输入种类的提示信息和验证信息也应该来自于那个配置文件。这里我们没有这么做，只是区分了密码和非密码字段。

一旦有了字段名列表，例程就迭代它来（向用户）询问字段的值，并且用这些获取的值来构造哈希条目。在交互会话结束的时候，对哈希的引用被返回来完成剩下的处理。下一步，我们要把这些信息写入队列。

在进一步介绍代码之前，让我们先讨论账户系统的数据存储和数据格式。

后端数据库

任何账户系统的核心都必然是某种数据库。有些管理员使用/etc/passwd文件或者SAM数据库（或者AD）来存放系统，但是这种尝试往往被证明是缺乏远见的。一旦开始使用自定义的数据库，我们就可以存放各种有用的信息，如账户的创建和过期时间、账户资质（是否是guest账户）、电话号码等等。而且未来可以用这个数据库做很多有用的事情，比如邮件列表自动创建等等。

为什么真正好的系统管理员都会创建账户系统

系统管理员往往分成两类：工程师和架构师。工程师喜欢在第一线了解细节信息。他们对系统软硬件状况的了解程度无人能及。如果系统的某个部分停止工作了，他们能立刻找到某条命令、某个文件，有时甚至能用扳手来修复。而某些有天分的工程师则更厉害，他们甚至能在机房的另外一头发现并修复出错的主机。

而架构师喜欢用自上而下的方式来分析系统。他们往往从更加抽象的角度来考虑系统的各个组成部分，并希望借此构造出更大、更复杂的系统。架构师感兴趣的是稳定性、可扩展性和可重用性。

从这两种人的身上，我们都能学到系统管理的技术。我最尊敬的系统管理员正是那种既能像工程师那样工作，又能像架构师那样思考的人。他们不仅仅是解决问题，而且会进行事后分析，从而找出避免问题再次发生的方法，这种态度能够带给他们长远的益处。

运行良好的计算系统需要构架师和工程师以共生关系工作。只有系统架构设计合理，工程师才能工作得更加高效。就汽车制造行业而言，车子出问题的时候，我们需要找工程师来修理，但是工程师也希望和设计精良的车打交道。架构师能设计出严谨的流水线、详细的服务手册和零部件的供应渠道，从而帮助工程师简单快速地完成任务。

这些角色在我们的案例中如何搭配呢？工程师往往会使用系统内置的工具来完成用户管理。他甚至已经有了一些自动完成管理任务（比如添加用户）的小脚本。而架构师则会从另外一个角度看问题，并且立刻开始构建账户系统。架构师会分析的问题有：

- 用户管理这个问题本身的重复性以及如何尽可能自动化。
- 账户系统需要采集的信息以及如何使用这些信息实现一些增值的功能。例如，可以在账户系统基础上构造出LDAP目录服务与自动网站生成工具。
- 如何保护账户系统中数据的安全。

- 如何在用户数量增长的情况下保持系统处理能力同步升级。
- 尽量使得其他地方也能使用这个系统。
- 其他的系统架构师是如何处理类似问题的。

“另外一个数据库”这个想法可能会让很多人紧张。他们会想：“大概我们需要花钱买一套商业软件，然后配置相应的机器，并且聘用数据库管理员。”其实如果系统真的有了成千上万的用户账户需要管理，那么这样的数据库可能会有必要。但是哪怕数据如此大量，你还是可以考虑使用非商业化的数据库系统（如PostgreSQL或者MySQL）来完成任务。你还可以参考第7章中关于使用Perl来管理这些数据库的详细介绍。

然而在这一章里面，我们对“数据库”的定义是非常广泛的。哪怕平面文件（flat-file）也可以说成是一个数据库，而且在数据量不大的情况下它也能胜任。Windows用户甚至可以使用Access数据库文件如`database.mdb`。^[注8]

为了提高可移植性并保持代码简洁，我们会使用非常酷的DBM::Deep模块，它的作者是Rob Kinyon。这个模块提供了让人吃惊的功能。它的自带文档的开始是这样写的：

一个使用平面文件的数据库模块，完全使用Perl开发。真正支持多级嵌套的哈希/数组（相比之下，MLDBM模块的实现是附加的），支持混合的OO/tie()接口，数据库文件跨平台FTP兼容，支持ACID事务，并且非常高效。可以处理上百万键-值对的存储以及无限层的嵌套，并且性能稳定。这不是一个依赖于C的DBM支持的模块，它完全从底层开始用Perl开发。完全兼容Unix、Mac OS X和Windows。

选择后端数据库格式

本书的第一版使用了XML文件作为后端数据库的存储格式，所以我觉得有必要声明选择数据库格式的依据，以便解除读者的困惑。上一版使用XML是因为它的格式相对简单，并且还有几个小小的益处，其中一个益处在于合乎技术的走向。早在2000年我就意识到，即使是系统管理员也应该学习一些处理XML的技巧。在之后的若干年，XML这个技术确实成为了系统管理中的一个重要技能。它在几个领域得到了长足应用，其中一个领域就是配置文件的存储。所以在后面的第6章我们还会深入介绍XML。而在这个案例中，XML并非天生适合用来支持数据库和队列建模，这就是为什么我要切换到另外一种数据库格式。

注8： 但是千万别用 Access 来实现一个多用户并发访问的数据库，那会带来潜在的灾难。

那么，什么是做这件事最好的数据库格式呢？这个问题的答案几乎总是在变化。从复杂性逐渐增长的次序来看，排在前面的技术应该是：

- 各种平面文件文本数据库（CSV、键-值对列表、YAML和XML等）
- DBM数据库（*ndbm*、*gdbm*、BerkeleyDB和DBM::Deep等）
- 基于文件的SQL数据库（如通过使用DBD::SQLite的DBI的SQLite）
- 需要服务器的SQL数据库（如DBI支持的MySQL、PostgreSQL、MS SQL或者Oracle）
- 某种类型的对象-关系映射器，或称ORM（如DBIx::Class、Rose::DB::Object和Jifty::DBI等）

列在最后的那一项（ORM）其实严格来说并不是一种数据库格式，它只是一种和数据打交道的方式。它能让你不必费力编写SQL，就能透明访问数据。它其实是在其他格式基础上的一种包装，而且这种包装往往能简化数据访问流程，所以我把它列在清单的后面。

选择哪种格式取决于你现在和将来的需要。我不能说哪种是正确的，因为数据量、读写并发度、可移植性等因素都制约着你的选择。这一章我们尽可能选择最简单（而不是相对简单）的技术。如果是设计一个生产系统，无疑我会选择某种基于SQL的数据库（并考虑使用ORM技术包装）。从贯彻本章的简单至上的角度出发，我选择了DBM::Deep，这是因为它避免了对SQL或者DBI的冗长介绍，并且在很多场合都可以使用。要想了解更多SQL相关的信息，请参考第7章。

如果你曾经用过Perl的哈希引用，那么使用DBM::Deep仿佛是在公园散步。例子如下：

```
use DBM::Deep;

my $db = DBM::Deep->new('accounts.db');

# 假定我们通过不断调用CollectInformation()已经获得了需要的数据结构：哈希的哈希

foreach my $user ( keys %users ) {

    # 还可以写成 $db->put($login => $users{$login});
    $db->{$login} = $users{$login};

}

# 稍后，在程序的其他部分或者另一个程序中可以……
```

```
foreach my $login ( keys %{ $db } ) {
    print $db->{$login}->{fullname}, "\n";
}
```

这里加粗的两行代码显示的语法，其实也就是标准Perl哈希引用语法。同时也表明可以存放哈希的哈希到数据库中，读取也同样方便。DBM::Deep还支持传统OOP调用，如同注释中显示的那样。

加入账户队列

让我们回到前面在“构建用户账户管理系统”一节引入的话题。我曾经提到要把CollectInformation()子例程采集到的账户信息存储在添加队列文件中。但我们并没有展示如何完成个任务。下面我们就用DBM::Deep模块来实现：

```
sub AppendAccount {
    use DBM::Deep;    # 马上会把这一行移到脚本的其他位置

    # 这里期望得到文件的全路径名
    my $filename = shift;

    # 这里期望得到一个匿名记录哈希
    my $record = shift;

    my $db = DBM::Deep->new($filename);

    $db->{ $record->{login} } = $record;
}
```

真的就这么简单。这个子例程只是简单使用了DBM::Deep的魔力哈希来加入另外一个键-值对而已。

注意： 有两件事情我得明说：

- 其实这并不是典型的队列，因为哈希的成员并不会以任何特定的顺序存储。如果这些真的让你觉得不便，可以考虑把哈希内容拿出来并按照记录修改时间（在CollectInformation()中采集到的字段）来排序，然后再开始处理。
 - 如果你两次输入同一个账户，那么后面输入的会覆盖前面那个，而这个特性在我们的案例中应该恰好是需要的。如果你要改变这样的覆盖也很容易，只要在写入哈希之前用exists()检查一下DBM::Deep数据结构里是否已经存在该键就可以了。请注意，这一章的代码特意设计成这么简单，所以将来在生产环境中，你很可能需要加入一些错误检查和商业逻辑验证的程序。
-

现在我们能在一行代码实现采集数据并写入到添加队列文件的操作：

```
AppendAccount( $addqueue, &CollectInformation );
```

从这个队列文件中读取信息也很容易，所有信息都在哈希里面了。所以我先不展示相关代码，直接在最后列出整个程序。

底层组件库

现在我们已经对数据有了掌控能力，不论是采集、读写还是存储都覆盖了，现在焦点应该转移到底层的账户系统。下一步让我们看看实际创建和删除用户的代码，这里要考虑的是如何设计可重用的组件。如果我们能把与账户系统相关的代码设计得更加合理，将来把它移植到其他操作系统就会更加容易。虽然移植这个问题看上去好像有点太遥远，但是系统管理生涯中唯一不变的就是变化。

Unix账户创建和删除的例程

让我们从创建Unix账户的代码开始。这里的代码非常简单，因为我们打算直接调用厂商提供的“add user”、“delete user”和“change password”可执行文件。

为什么用这样“稀松平常”的方式呢？这是因为我们知道，厂商提供的可执行文件往往能和环境中的各种组件“和平共处”。比如说：

- 自动加锁（这样不必担心两个程序同时修改密码文件而导致文件损坏）。
- 自动处理前面讨论过的各种密码文件格式的变种（也能处理好密码的编码问题）。
- 更好地识别各种操作系统的验证方案或者分布式密码存储机制。比如，我发现在某个Unix变种上，外部的“add user”可执行文件能自动把用户添加到NIS中去。

使用外部二进制程序来创建和删除账户的缺点在于：

OS变种

每个操作系统都有相似的用户管理程序，这些程序往往存储在不同的路径，使用略微不同的参数。从兼容性角度来看，几乎所有的Unix变种（包括Linux，不包括BSD）都有`useradd`和`userdel`这两个程序可用于添加和删除账户。而BSD分支大都使用`adduser`和`rmuser`两个程序来做相同的事情，只是命令行参数有很大差别。这样的差异对我们的系统来说是个麻烦问题。^[注9]如今有些人已经开始对管理命令进行规范化（如POSIX就是这样一种标准），但是从实践角度来看还没有办法依赖任何约定。

注9： 另外，如果你真的对变种间的差异感兴趣，就去看看 OS X 吧。到目前为止，它还没有像样的专用账户管理命令。你必须学习使用 `dscl`，它和 `NetInfo` 一脉相承。这里有人和我一样钟情于 NeXT 这样的老古董吗？

单机限制

大多数命令行工具只能在本地机器上工作。如果你的用户账户信息存储在中心系统（如现今最值得提倡的LDAP），那么这些命令往往不知如何创建用户。不过，Windows的net命令是个例外：常常见到用户使用扩展的user*命令来完成这些任务，有些人甚至是用Perl来实现。

安全因素

这里调用的程序和传递的参数会被那些不断执行ps命令的用户捕捉到。但是如果账户只在安全的服务器上创建，信息泄露就能有效避免。

附加的依赖性

如果可执行程序在某种特殊情况下被移动了，或者有所改变，那么我们的账户系统就玩不转了。

失去控制

我们必须留意账户创建的过程是原子的（atomic）。也就是说必须牢记，外部程序一旦启动我们就无法介入，不可以插入我们自己的操作。这使得错误检查和修复变得更加困难。

难以一次完成整个任务

使用这些程序常常不能一次完成所有必要的步骤。如果你需要把用户加入某些附加组，或者把用户加入某个特定邮件列表，或者把用户加入某个商业软件包的license文件，你必须添加更多代码来处理这些特殊的需求。这并非是这个方式特有的问题，而是一种常见问题：几乎任何账户管理系统都不可能把全部功能建立在一两条外部命令上。这一点对于系统管理员来说应该不奇怪，因为系统管理这份工作大多数时候并不是“闲庭漫步”。

从账户管理系统演示的角度来看，此方式的优点多于缺点，所以我们最终还是决定通过调用外部可执行程序来实现此功能。为了简化问题，我们只考虑基于Linux的单机系统，忽略那些复杂的NIS或者BSD系统。如果你觉得这个假定有点太过简化，可以考虑使用Randy Maas创作的CfgTie系列模块进行扩展。在展示了Linux版的代码之后，我们会回头看看那些命令行管理界面不够友善的Unix变种应该吸取什么教训。

下面就是最基础的账户创建例程：

```
# 这些变量应该通过配置文件来设置
Readonly my $useraddex => '/usr/sbin/useradd'; # useradd 命令的位置
Readonly my $homeUnixdirs => '/home';          # home 目录的根目录
Readonly my $skeldir      => '/home/skel';      # 原型的home目录
Readonly my $defshell     => '/bin/zsh';        # 默认 shell

sub CreateUnixAccount {
```

```

my ( $account, $record ) = @_ ;

### 构造命令行，使用：
# -c = 注释字段
# -d = home 目录
# -g = 组名（默认为用户类型）
# -m = 创建 home 目录
# -k = 拷贝主干目录中的文件
# -p = 设置密码
# （另外还可以用 -G 组1, 组2, 组3 参数来设置附属组）
my @cmd = (
    $useraddex,
    '-c', $record->{fullname},
    '-d', "$homeUnixdirs/$account",
    '-g', $record->{type},
    '-m',
    '-k', $skeldir,
    '-s', $defshell,
    '-p', $record->{password},
    $account
);

# 这个变量会获得调用@cmd命令的shell返回值，而非system()的返回值

my $result = 0xff & system @cmd;

# 返回零意味着成功，返回非零意味着出错，所以我们需要反转
return ( ($result) ? 0 : 1 );
}

```

这段代码能把适当条目加入密码文件，为账户创建home目录，并且从主干目录复制默认环境文件（*.profile*、*.tcshrc*和*.zshrc*等）。

对应的是删除账户的代码，这相比之前的创建来说更简单些：

```

# 这个变量其实应该通过配置文件来设置
Readonly my $userdelex => '/usr/sbin/userdel'; # userdel 命令的位置
sub DeleteUnixAccount {

    my ( $account, $record ) = @_ ;

    ### 构造命令行，使用：
    # -r = 删除账户的 home 目录
    my @cmd = ( $userdelex, '-r', $account );

    my $result = 0xff & system @cmd;

    # 返回零意味着成功，返回非零意味着出错，所以我们需要反转
    return ( ($result) ? 0 : 1 );
}

```


Unix账户创建和删除例程——延伸

在进入关于Windows的讨论之前，让我们先看看刚才的代码在Unix的某个衍生系统上的运行情况，这会有助于我们理解兼容性问题。这里我不仅会展示那些非常酷的技术，而且还会指出系统中微不足道的差异会导致多大的开发问题。

一个折磨人的细微差异是Solaris的`useradd`命令无法通过`-p`开关来设置账户的哈希密码。其实它也有`-p`开关，不过和Linux里面的用处不一样。看到这里你大概会想，只要稍微修改一下`CreateUnixAccount()`里面的代码，将`@cmd`设置为反映Solaris为此目的而使用的命令行参数就好了。不过只要开始查找Solaris的手册里面关于`useradd`的帮助信息，你就会被弄得一头雾水，然后很快会得出结论：Solaris不支持在创建账户的同时设置哈希密码，只能在创建账户之后立刻设置新密码（其实只有设置密码才能解开在新建账户时自动加上的锁）。

这会给代码带来许多影响。首先我们要给`CreateUnixAccount()`加入一些东西，这样就能在创建一个账户之后立即设置密码。这并不难，需要加入的代码可能是这样：

```
$result = InitUnixPasswd( $account, $record->{'password'} ) );  
return 0 if (!$result);
```

然后再编写一段 `InitUnixPasswd()` 例程。不过这还不算是代码中最重要的修改。最大的问题在于我们现在必须在队列里面存放明文密码，因为哈希过的密码是不可逆的（也就是不可解密），用户无法修改密码。现在你大概开始明白`CollectInformation()`的代码里面为什么有这样奇怪的注释了吧：

```
# 如果遇到密码字段，使用随机的 salt 值对它进行加密，然后存储  
if ( $field eq 'password' ) {  
  
    # 请相信我，稍后我们会对存储哈希版本密码的决定后悔的，  
    # 本章稍后会揭晓后悔的原因  
    $record{$field} = unix_md5_crypt(  
        prompt( 'p', 'Please enter password:', '', '' ), undef );  
    }  
}
```

这正是让我们后悔的决定。稍后创建Windows账户的时候我们还要后悔一次，因为那里也必须使用明文密码。我不会在这里展示解决方案，或许最好的折中做法是从`Crypt::`命名空间中挑出一个密码模块用来进行加密、解密。^[注10]提到这些是为了表明那些细微的差别往往会让程序开发进入两难的困境。

注10：这意味着保护密码安全重任现在落在了脚本或者脚本的相关配置文件的上面。也就是说，我们必须给明文密码进行额外加密、解密，还必须确保程序和数据本身的安全。而这些问题我们目前无暇顾及。

在修改了密码提示和存储的代码之后，下一步就是编写修改密码的代码了。但这时候你会发现Solaris没有提供任何非交互式修改密码的程序，这会让人为之一震。^[注11]不过我们还是可以通过某种花招来设置密码，所以我们把它用子例程封装起来，这样使用起来会容易些。

Solaris手册里面说：“新创建的账户保持锁定状态，直到passwd(1)命令被执行才会解开。”passwd <accountname> 命令会给那个账户设置密码，这看上去很简单，但其实潜伏着一个问题。passwd 命令期望的是真的用户输入，而且它也花了相当多的力气直接读取终端设备。结果下面的尝试会失败：

```
# 以下代码不会正常工作
open my $PW, "|passwd $account";
print $PW $newpasswd, "\n";
print $PW $newpasswd, "\n";
close $PW;
```

我们必须使用更加特殊的方法来“诱使”passwd认为输入密码的是人，而非Perl程序。要达到这个目的可以使用Expect模块，作者是Austin Schutz，目前的维护者是Roland Giersig，它能在一个虚拟终端里面与另外一个程序进行交互。Expect依赖于著名的Tcl程序Expect（作者是Don Libes）。它属于程序间双向交互模块的大家庭，我们会第9章看到它的兄弟：Jay Rogers写的Net::Telnet模块。

这些模块都有类似的运行模式：等待一个程序的输出，然后发送一些输入回应信息，等待回应，再发送些数据，诸如此类。下面的代码能在虚拟终端里面启动passwd，等待它的命令行提示。然后我们的程序和passwd之间的交谈应该不难理解：

```
Readonly my $passwdex => '/usr/bin/passwd'; # passwd 命令的位置
sub InitUnixPasswd {
    use Expect; # 稍后我们会把它移到其他位置

    my ( $account, $passwd ) = @_ ;

    # 期待返回一个进程对象
    my $pobj = Expect->spawn( $passwdex, $account );
    die "Unable to spawn $passwdex: ${!}\n" unless ( defined $pobj );

    # 封住标准输出（也就是让程序保持安静）
    $pobj->log_stdout(0);

    # 等待输入密码（或再次输入）的提示信息，并自动应答
    $pobj->expect( 10, 'New password: ' );

    # Linux 上有时候提示符出现得过早，所以我们等待1秒再发送输入提示
```

注11：如果你并不介意使用那些不随 Solaris 发行的软件包，在修改密码时还可以使用 *changepass* 命令。这个命令是 *cgipaf* 软件包提供的，此软件包可在 <http://www.wagemakers.be/english/programs/cgipaf> 获得。

```

sleep 1;
print $obj "$passwd\r";
$obj->expect( 10, 'Re-enter new password: ' );
print $obj "$passwd\r";

# 密码是否被接受?
$result
    = ( defined( $obj->expect( 10, 'successfully changed' ) ) ? 1:0 );

# 关闭进程对象，给程序 15 秒时间退出
$obj->soft_close();

return $result;
}

```

这里我们粗浅地使用了Expect模块，而它其实能完成更加复杂的操作。请参考这个模块自带的文档了解更多信息。

这里我们还需要介绍另外一种解决方案。我不太喜欢这种方法，因为它跳过了某些密码修改的步骤。但是它应该也有一些用处，因为你可能不喜欢对passwd进行脚本化。这种情况下请使用Eric Estabrook的Passwd::Solaris模块，直接对Solaris的/etc/passwd和/etc/shadow文件进行修改。这个模块支持哈希密码的输入。

警告： 如果你想直接在passwd和shadow文件中填写哈希密码，请确保你的Solaris版本高于第9版（12/02发行），并且在/etc/security/policy.conf里面已经设置了对于哈希算法的兼容选项。

Windows账户创建和删除例程

对于Windows来说，创建和删除账户的步骤要比Unix上简单一些，因为可以通过标准API调用来完成。我们可以使用外部可执行程序（如随处可见的net命令，加上其开关USERS/ADD），同时也可以使用之前提到过的很多模块直接进行API调用。可以使用的模块包括Win32::NetAdmin、Win32::UserAdmin、Win32API::Net和Win32::Lanman等等。而Active Directory的用户则可以通过第9章会介绍的ADSI调用来完成此任务。

从这些Windows模块中进行筛选时更多的是看个人喜好。为了展示它们之间的区别，我们可以看看它们背后使用的API调用，可以在<http://msdn.microsoft.com>上的Network Management SDK 文档找到这些调用（如果你觉得找文档很困难，请搜索“NetUserAdd”关键字）。NetUserAdd()和其他的调用都有一个说明信息级别的参数。比如在使用信息级别1的时候，传送的C语言数据结构是这样的：

```

typedef struct _USER_INFO_1 {
    LPWSTR    usr11_name;
    LPWSTR    usr11_password;
    DWORD     usr11_password_age;
}

```

```

DWORD    usri1_priv;
LPWSTR   usri1_home_dir;
LPWSTR   usri1_comment;
DWORD    usri1_flags;
LPWSTR   usri1_script_path;
}

```

而信息级别是2的时候，数据结构就相应地扩充了一些：

```

typedef struct _USER_INFO_2 {
    LPWSTR    usri2_name;
    LPWSTR    usri2_password;
    DWORD     usri2_password_age;
    DWORD     usri2_priv;
    LPWSTR    usri2_home_dir;
    LPWSTR    usri2_comment;
    DWORD     usri2_flags;
    LPWSTR    usri2_script_path;
    DWORD     usri2_auth_flags;
    LPWSTR    usri2_full_name;
    LPWSTR    usri2_usr_comment;
    LPWSTR    usri2_parms;
    LPWSTR    usri2_workstations;
    DWORD     usri2_last_logon;
    DWORD     usri2_last_logoff;
    DWORD     usri2_acct_expires;
    DWORD     usri2_max_storage;
    DWORD     usri2_units_per_week;
    PBYTE     usri2_logon_hours;
    DWORD     usri2_bad_pw_count;
    DWORD     usri2_num_logons;
    LPWSTR    usri2_logon_server;
    DWORD     usri2_country_code;
    DWORD     usri2_code_page;
}

```

第3级和第4级（第4级才是微软推荐使用的格式^[注12]）的数据结构如下所示：

```

typedef struct _USER_INFO_3 {
    LPWSTR usri3_name;
    LPWSTR usri3_password;
    DWORD usri3_password_age;
    DWORD usri3_priv;
    LPWSTR usri3_home_dir;
    LPWSTR usri3_comment;
    DWORD usri3_flags;
    LPWSTR usri3_script_path;
    DWORD usri3_auth_flags;
    LPWSTR usri3_full_name;
}

```

注12：如果在这里继续展示更高级别的数据结构就有些折磨人了，因为到目前为止还没有哪个 Perl 模块可以支持。第3级和第4级的结构大致相同，只是更加复杂一些，相信你马上就会看到。

```

LPWSTR usri3_usr_comment;
LPWSTR usri3_parms;
LPWSTR usri3_workstations;
DWORD usri3_last_logon;
DWORD usri3_last_logoff;
DWORD usri3_acct_expires;
DWORD usri3_max_storage;
DWORD usri3_units_per_week;
PBYTE usri3_logon_hours;
DWORD usri3_bad_pw_count;
DWORD usri3_num_logons;
LPWSTR usri3_logon_server;
DWORD usri3_country_code;
DWORD usri3_code_page;
DWORD usri3_user_id;
DWORD usri3_primary_group_id;
LPWSTR usri3_profile;
LPWSTR usri3_home_dir_drive;
DWORD usri3_password_expired;
}

```

以及：

```

typedef struct _USER_INFO_4 {
    LPWSTR usri4_name;
    LPWSTR usri4_password;
    DWORD usri4_password_age;
    DWORD usri4_priv;
    LPWSTR usri4_home_dir;
    LPWSTR usri4_comment;
    DWORD usri4_flags;
    LPWSTR usri4_script_path;
    DWORD usri4_auth_flags;
    LPWSTR usri4_full_name;
    LPWSTR usri4_usr_comment;
    LPWSTR usri4_parms;
    LPWSTR usri4_workstations;
    DWORD usri4_last_logon;
    DWORD usri4_last_logoff;
    DWORD usri4_acct_expires;
    DWORD usri4_max_storage;
    DWORD usri4_units_per_week;
    PBYTE usri4_logon_hours;
    DWORD usri4_bad_pw_count;
    DWORD usri4_num_logons;
    LPWSTR usri4_logon_server;
    DWORD usri4_country_code;
    DWORD usri4_code_page;
    PSID usri4_user_sid;
    DWORD usri4_primary_group_id;
    LPWSTR usri4_profile;
    LPWSTR usri4_home_dir_drive;
    DWORD usri4_password_expired;
}

```

你不必懂得这些参数，甚至不必懂得C语言，也能看出在创建用户的时候必须提供的信息是随着级别的上升而增加了。而且每个级别都是前一级别的超集（superset）。

那么这对 Perl 来说有什么影响呢？我之前提到的模块都必须作出的决策有：

- 这个“信息级别”概念是否需要让 Perl 程序员知道呢？
- 程序员需要提供哪个级别的信息（也就是多少参数）来创建用户呢？

Win32API::Net 和 Win32::UserAdmin 都允许程序员选择信息级别，Win32::NetAdmin 和 Win32::Lanman 则不支持。在这些模块中，Win32::NetAdmin 支持的参数最少，如 full_name 字段就不能在用户创建的时候设置。所以，如果你选择使用这个模块，却希望设置更多参数，那么必须再调用其他模块。

现在你大概明白为什么我说模块的选择更多是看个人喜好了。一个好的筛选方法是：先确定那些重要的参数，并且把这些参数存储在数据库中，然后找一个合适的模块来兼容这些参数。为了满足这里展示的需要，我们选择了 Win32API::Net，这样可以和前面的例子保持兼容。下面就是为账户系统创建用户和删除用户的代码：

```
use Win32API::Net qw(:User :LocalGroup);    # 用来创建账户
use Win32::Security::NamedObject;          # 用来设置 home 目录属性
use Readonly;

# 除 home 目录之外，还会给每个用户创建一个“data dir”
# （在用户第一次登录操作系统的时候，系统会自动创建 home 目录）
Readonly my $homeWindirs => '\\\\homeserver\\home'; # home directory root dir
Readonly my $dataWindirs => '\\\\homeserver\\data'; # data directory root dir

sub CreateWinAccount {

    my ( $account, $record ) = @_ ;

    my $error;    # 稍后用来返回可能的错误信息

    # 更加理想的情况是从数据库中获取这些默认值
    my $result = UserAdd(
        '',          # 在本机创建账户
        3,          # 这里指定了 USER_INFO_3 级别的信息详细程度
        {
            acctExpires => ?1,    # 账户永不过期
            authFlags    => 0,    # 只读，不必设置
            badPwCount   => 0,    # 只读，不必设置
            codePage     => 0,    # 默认值
            comment      => '',    # 不必设置，用户自行更改
            countryCode => 0,    # 默认值
            # 普通账户必须设置的标志位
            flags        => (
                Win32API::Net::UF_SCRIPT() & Win32API::Net::UF_NORMAL_ACCOUNT()
            ),
            fullName => $record->{fullname},
        }
    );
```

```

homeDir => "$homeWindirs\\$account",
homeDirDrive => 'H',          # 用 H: 作为 home 目录所在的盘符
lastLogon => 0,               # 只读, 不必设置
lastLogoff => 0,              # 只读, 不必设置
logonHours => [],             # 没有登录时间限制
logonServer => '',            # 只读, 不必设置
maxStorage => ?1,             # 没有磁盘限额
name => $account,
numLogons => 0,                # 只读, 不必设置
parms => '',                  # 不必设置
password => $record->{password}, # 明文密码
passwordAge => 0,              # 只读
passwordExpired =>
    0, # 用户首次登录之后不必重设密码
primaryGroupId => 0x201,      # 说明文档中建议的值, 具体原因不详
priv => USER_PRIV_USER(),    # 非管理员账户
profile => '',                 # 目前不必设置
scriptPath => '',             # 没有登录时执行的脚本
unitsPerWeek => 0,            # 用来计算 logonHours, 这里不必设置
usrComment => '',             # 不必设置, 让用户自行更改
workstations => '',           # 不必指定工作站的数量
userId => 0,                  # 只读
},
$error
);

return 0 unless ($result); # 也可以返回 Win32::GetLastError()

# 把账户加入合适的 LOCAL 组
# 这里假定组名就是账户名
$result = LocalGroupAddMembers( '', $record->{type}, [$account] );
return 0 if (!$result);

# 创建 data 目录
mkdir "$dataWindirs\\$account", 0777
    or (warn "Unable to make datadir:$!" && return 0);

# 设置目录的属性
my $datadir = Win32::Security::NamedObject->new( 'FILE',
    "$dataWindirs\\$account" );
eval { $datadir->ownerTrustee($account) };
if ($?) {
    warn "can't set owner: $?";
    return 0;
}

# 授予用户对目录和其下所有文件的控制权
my $dacl
    = Win32::Security::ACL->new( 'FILE',
        [ 'ALLOW', 'FULL_INHERIT', 'FULL', $account ],
    );

eval { $datadir->dacl($dacl) };
if ($?) {
    warn "can't set permissions: $?";
}

```

```

        return 0;
    }
}

```

而用户删除代码则是这样：

```

use Win32API::Net qw(:User :LocalGroup);    # 用来删除账户
use File::Path 'remove_tree';              # 用来递归删除目录
use Readonly;

sub DeleteWinAccount {

    my ( $account, $record ) = @_;

    # 只从 LOCAL 组删除用户账户。如果我们还打算删除全局组，
    # 那么可以把下面两个 Win32API::Net 调用中的 "Local"
    # 删掉（例如 UserGetGroups 和 GroupDelUser）
    # 另外：UserGetGroups 可以带上一个标志，从而进行间接组成员查询
    #       （例如：某个用户属于组 A，而组 A 又属于组 B，
    #       最终用户也间接属于组 B）
    UserGetLocalGroups( '', $account, \my @groups );
    foreach my $group ( @groups ) {
        return 0 if ( ! LocalGroupDelMembers( '', $group, [$account] ) );
    }

    # 从本地机器上删除该用户账户
    # （即，清空第一个参数）
    unless ( UserDel( '', $account ) ) {
        warn 'Can't delete user: ' . Win32::GetLastError();
        return 0;
    }

    # 删除 home 和 data 目录，包括其下的文件和子目录
    # remove_tree 会把错误信息存储在 $err 中，它是对一个哈希引用的数组的引用
    # 注意：remove_tree() 是 File::Path 2.06 版本之后引入的，之前的函数叫做 rmtree
    remove_tree( "$homeWindirs\\$account", { error => \my $err } );
    if (@$err) {
        warn "can't delete $homeWindirs\\$account\n" ;
        return 0;
    }

    remove_tree( "$dataWindirs\\$account", { error => \my $err } );
    if (@$err) {
        warn "can't delete $dataWindirs\\$account\n" ;
        return 0;
    }
    else {
        return 1;
    }
}

```

顺便说一下，以上代码使用了可移植的 File::Path 模块来完成账户的 home 目录的删除。如果我们想做些更加专业的操作，如把这个目录放入回收站，那么可以尝试使用

Win32::FileOp 模块，它的作者是 Jenda Krynický，可以在 <http://jenda.krynicky.cz> 找到。在这种情况下，你可以载入这个模块，并且把 `rmtree()` 那一行改成这样：

```
# 将会把目录放入回收站，另外还可以考虑先获得用户的确认
# 这种确认机制需要事先在账户系统中进行相应的设置
my $result = Recycle("$homeWindirs\\$account");
my $result = Recycle("$dataWindirs\\$account");
```

这个模块还有一个 `Delete()` 函数，可以用来完成和 `remove_tree()` 调用同样的操作，而且更加快些（不过兼容性要差些）。

处理脚本

一旦我们有了后端数据库，就可以开始编写那些用于完成日常用户管理的脚本。这些脚本会基于一个底层的组件库 (*Account.pm*)，而它其实是汇集了之前编写的所有子例程而形成的库文件。要让它能以模块的方式加载，可以在末尾加上 `1`；我们会在此做一些改进：把模块和变量初始化的代码都放到公共的初始化子例程中，删除原来子例程中的初始化代码（只留下必要的 `our` 语句）。下面就是初始化子例程的代码：

```
sub InitAccount {

    # 无论 Linux 还是 Win32 例程都会用到这些模块
    use DBM::Deep;
    use Readonly;
    use Term::Prompt;

    # 无论 Linux 还是 Win32 例程都需要这些全局变量
    Readonly our $record =>
        { fields => [ 'login', 'fullname', 'id', 'type', 'password' ] };
    Readonly our $addqueue => 'add.db'; # 添加账户队列文件的名称
    Readonly our $delqueue => 'del.db'; # 删除账户队列文件的名称
    Readonly our $maindata => 'acct.db'; # 主账户数据库文件的名称

    # 载入 Win32 特有的模块并设置 Win32 特有的全局变量
    if ( $^O eq 'MSWin32' ) {
        require Win32API::Net;
        import Win32API::Net qw(:User :LocalGroup);
        require Win32::Security::NamedObject;
        require File::Path;
        import File::Path 'remove_tree';

        # 账户文件的位置
        Readonly our $accountdir => "\\server\accounts\system\\";

        # 邮件列表，稍后会有范例使用它
        Readonly our $maillists => $accountdir . "maillists\\";

        # home 目录的根目录
        Readonly our $homeWindirs => "\\homeserver\home";
```

```

# data 目录的根目录
Readonly our $dataWindirs => "\\homeserver\home";

# 账户添加子例程的名字
Readonly our $accountadd => \&CreateWinAccount;

# 账户删除子例程的名字
Readonly our $accountdel => \&DeleteWinAccount;
}

# 载入 Unix 特有的模块并设置 Unix 特有的全局变量
else {
    require Expect; # 为 Solaris 密码设置的需要而载入
    require Crypt::PasswdMD5;

    # 账户文件的位置
    Readonly our $accountdir => '/usr/accountsystem/';

    # 邮件列表, 稍后会有范例使用它
    Readonly our $maillists => '$accountdir/maillists/';

    # 用户添加可执行程序的位置
    Readonly our $useraddex => '/usr/sbin/useradd';

    # 删除用户可执行程序的位置
    Readonly our $userdelex => '/usr/sbin/userdel';

    # 密码设置可执行程序的位置
    Readonly our $passwdex => '/usr/bin/passwd';

    # home 目录的根目录
    Readonly our $homeUnixdirs => '/home';

    # 原型的home 目录
    Readonly our $skeldir => '/home/skel';

    # 默认 shell
    Readonly our $defshell => '/bin/zsh';

    # 账户添加子例程的名字
    Readonly our $accountadd => \&CreateUnixAccount;

    # 账户删除子例程的名字
    Readonly our $accountdel => \&DeleteUnixAccount;
}
}

```

让我们来看看使用此模块的示范脚本。下面是把添加账户的任务存入队列的脚本：

```

use Account;

# 载入我们的底层库例程
&InitAccount;

```

```

# 读取添加账户“队列”的内容
my $queue = ReadAddQueue();

# 尝试创建队列中所有的账户
ProcessAddQueue($queue);

# 把创建好的账户记录到主数据库中
# 如果创建过程中出了问题，就把此记录退回队列
DisposeAddQueue($queue);

# 把添加账户队列的内容读入到 $queue 数据结构中
sub ReadAddQueue {

    our ( $accountdir, $addqueue );
    my $db = DBM::Deep->new( $accountdir . $addqueue );

    my $queue = $db->export();

    return $queue;
}

# 遍历队列中的所有创建请求，为其中每个请求（也就是每个键）完成创建账户操作

sub ProcessAddQueue {
    my $queue = shift;

    our $accountadd;
    foreach my $login ( keys %{$queue} ) {
        my $result = $accountadd->( $login, $queue->{$login} );
        if ( $result ) {
            $queue->{$login}{status} = 'created';
        }
        else {
            $queue->{$login}{status} = 'error';
        }
    }
}

# 现在再次遍历队列，把那些状态是“created”的账户附加至主数据库，
# 把其他的账户重新写入添加队列，并覆盖原始记录。
sub DisposeAddQueue {
    my $queue = shift;

    our ( $accountdir, $addqueue, $maindata );

    my $db = DBM::Deep->new( $accountdir . $addqueue );

    foreach my $login ( keys %{$queue} ) {
        if ( $queue->{$login}{status} eq 'created' ) {
            $queue->{$login}{login} = $login;
            $queue->{$login}{creation_date} = time;
            AppendAccount( $accountdir . $maindata, $queue->{$login} );
            delete $queue->{$login};      # 从内存镜像中删除
            delete $db->{$login};        # 从磁盘数据库文件中删除
        }
    }
}

```

```

    }
}

# 此时 $queue 中剩下的就是那些创建失败的账户

# 把它们重新并入队列
my $db = DBM::Deep->new( $accountdir . $addqueue );

my $queue = $db->import($queue);
}

```

下面再来看看处理删除账户队列的脚本，它和之前的代码有相似的结构：

```

use Account;

# 载入我们的底层库例程
&InitAccount;

# 读取删除账户“队列”的内容
my $queue = ReadDelQueue();

# 尝试删除队列中所有的账户
ProcessDelQueue($queue);

# 把成功删除的账户记录到主数据库中
# 如果删除过程中出了问题，就把此记录退回队列
DisposeDelQueue($queue);

# 把删除账户队列的内容读入到 $queue 数据结构中
sub ReadDelQueue {

    our ( $accountdir, $delqueue );
    my $db = DBM::Deep->new( $accountdir . $delqueue );

    my $queue = $db->export();

    return $queue;
}

# 遍历队列中的所有删除请求，为其中每个请求（也就是每个键）完成删除账户操作
sub ProcessDelQueue {
    my $queue = shift;

    our $accountdel;
    foreach my $login ( keys %{$queue} ) {
        my $result = $accountdel->( $login, $queue->{$login} );
        if ( !defined $result ) {
            $queue->{$login}{status} = 'deleted';
        }
        else {
            $queue->{$login}{status} = 'error';
        }
    }
}
}

```

```

# 现在再次遍历队列，为那些状态是“deleted”的账户更改主数据库信息。
# 把所有无法删除的账户合并回账户删除队列，更新。
sub DisposeDelQueue {
    my $queue = shift;

    our ( $accountdir, $delqueue, $maindata );

    my $maindata = DBM::Deep->new( $accountdir . $maindata );
    my $delqueue = DBM::Deep->new( $accountdir . $delqueue );

    foreach my $login ( keys %{$queue} ) {
        if ( $queue->{$login}{status} eq 'deleted' ) {
            $maindata->{$login}{deletion_date} = time;
            delete $queue->{$login}; # 从内存镜像中删除
            delete $delqueue->{$login}; # 从队列上的删除队列文件中删除
        }
    }

    # 此时 $queue 中剩下的就是那些删除失败的账户。
    # 我们把它们重新并入删除队列
    # 以便将来进一步处理

    $delqueue->import($queue);
}

```

你可能会想到更多有用的处理脚本。比如我们可以使用脚本来执行数据导出和一致性检查任务（如验证用户的home目录是否与主数据库中的账户类型匹配、用户所属的组是否正确？）限于篇幅，我们无法在这里一一讨论各种可能的处理，只能展示一个数据导出的例子。之前曾经提到过，可以用这个数据库来实现按用户类型区分的邮件列表。下面的代码能读取主数据库并创建一个包含用户名的文件集合，每个用户类型对应一个文件：

```

use Account; # 这只是为了获取文件的位置

&InitAccount;

# 读取所有信息，这个操作在数据量大的时候会有问题
my $database = ReadMainDatabase();

WriteFiles($database);

# 把主数据库读入哈希的哈希中
sub ReadMainDatabase {
    our ( $accountdir, $maindata );
    my $db = DBM::Deep->new( $accountdir . $maindata );

    my $database = $db->export();

    return $database;
}

# 遍历所有的键，编译某一类型账户的列表，

```

```

# 并把它们存入一个列表的哈希中。
# 然后把每个键的内容写出至一个不同的文件
sub WriteFiles {

    my $database = shift;

    our ( $accountdir, $maillists );

    my %types;

    foreach my $account ( keys %{$database} ) {
        next if $database->{$account}{status} eq 'deleted';
        push( @{$types{$database->{$account}{type}} }, $account );
    }

    foreach my $type ( keys %types ) {
        open my $OUT, '>', $maillists . $type
            or die 'Unable to write to ' . $maillists . $type . ": $!\n";
        print $OUT join( "\n", sort @{$types{$type}} ) . "\n";
        close $OUT;
    }
}

```

如果查看邮件列表目录，就能看到：

```

> dir
faculty  staff

```

这些文件里的每一个都包含了适当的用户账户列表。

账户系统总结

我们已经展示了账户系统的四个组件，是时候总结一下，看看还有哪些缺失的功能（除了那些过于细致的功能）：

错误检查

我们的样本程序只有最基本的错误检查代码。任何成熟的账户系统都需要加入额外40%~50%的错误检查代码，用来排除那些数据和系统交互中的错误。

错误报告

为了保持简洁，这里的代码只是做了最少的错误报告，并不足以帮助我们调试问题。例程在错误发生时只是简单地返回0，其实应该返回异常信息（或者异常对象）来反映问题本质。我们可以从系统中获得更详细的信息，如在Win32API::Net调用之后，可以用Win32::GetLastError()（或者更进一步地用Win32::FormatMessage(Win32::GetLastError())这个最佳组合）来报告Windows中的错误信息。

面向对象

尽管我自己并不是那种热衷于面向对象的程序员，但我还是不想看到散落在代码中

的那些全局变量。你可以尝试把代码改写成面向对象的版本，但我并没有指望通过这个例子来推动你学习OOP。

可扩展性

我们的代码在小型或者中型环境中能正常工作，但所有写着“读入整个文件到内存”的地方都有可能出现问题。要增强可扩展性，至少需要改进我们的数据存储和读取技术。

安全性

这是和之前提到的错误检查相关的。除了那些明显的安全漏洞之外（如存储明文密码），我们没有在代码中引入必要的安全检查，没有验证数据源（如队列文件）的安全性。至少需要再加入20%~30%的代码来完成这个任务。

多用户

我们的代码中最大的漏洞应该就是这个了。我们没有确保代码在多用户或者多脚本同时运行的情况下可以正常运作。就理论而言DBM::Deep能自动处理文件上锁，但我们并没有完全证明这一点。这个问题非常重要，所以我们会在这一节的后面部分再次讨论它。

可维护性

考虑到这些问题的存在，我们必须加入不少代码来改进。而且将来可能还有更多功能需要加入，代码的复杂性肯定会上升。这最终会导致一个带有对业务而言关键的功能的大型、复杂、跨平台的程序浮出水面。那么我们所在的机构是否有人力来支持它的运行呢，或是简单地把维护责任丢给原始开发人员呢？这些问题必须得到妥善解决。

解决多用户问题的一个方法就是引入有效的文件锁机制。文件锁使得多个脚本能够和谐共处。在某个脚本读写文件之前，它必须先尝试锁住文件。只要能获得锁，那么文件的读写就是安全的。反之，如果其他脚本正在使用文件，那么这个脚本就不能锁住文件，于是它就知道最好不要对文件做什么操作（否则会导致文件损坏）。当然，从更加广义的角度来说，锁机制和多用户读写并非这么简单（查看操作系统或分布式系统文档就能明白我在说什么）。比如，在网络文件系统中实施文件锁就非常困难。DBM::Deep的说明文档就明确说明了文件锁在NFS文件系统上会无法工作。如果你不信任这个模块内置的锁机制，那么请考虑下面列出的那些针对Perl的建议：

- 有些小技巧能得到意想不到的效果。我喜欢的方法是使用著名的邮件过滤程序 *procmail* (<http://www.procmail.org>) 自带的 *lockfile* 程序。*procmail* 安装的时候会尽全力来判断在你的文件系统中如何实施安全锁机制。而 *lockfile* 则能够使用这个成果，

并且避免不必要的复杂性。

如果你不想使用外部可执行程序，那么还可以考虑众多的文件锁模块，如David Muir Sharnoff的File::Flock、Tom Christiansen和Nathan Torkington合著的《Perl Cookbook》一书里面介绍的File::LockDir模块、Kenneth Albanowski的File::Lock、Paul Henson的File::Lockf、Raphael Manfredi的Lockfile::Simple。它们的区别通常只是调用接口不同，不过Lockfile::Simple尝试绕过Perl的flock()函数来实现锁。比较一下，找出到底哪个模块是你需要的。

只要记住在修改数据之前上锁（或者在读取可能改变的数据之前上锁），并且确保在修改完之后（如文件关闭后）才解锁，那么文件锁机制并不复杂。要了解这方面的其他信息，请参考《Perl Cookbook》、Perl 常见问题解答(Frequently Asked Questions, FAQ)列表、flock()函数以及DB_File模块的Perl文档。

到这里，我们关于账户管理的讨论就结束了，我们同时还介绍了如何从架构师的角度来提升它的能力。关于这个系统的讨论（尤其是那些对潜在问题的分析）对我们将来的系统管理项目非常有益，而且这些理念已经超越了编写脚本的范畴。

在这一章中，我们关注了账户生命周期的创建和消亡两个时间点。在下一章，我们会关心用户在这两个时间点之间所做的事情。

本章所用模块

模块名	CPAN ID	版本
User::pwent（随 Perl 发布）		1.00
File::stat（随 Perl 发布）	1.01	
Passwd::Solaris	EESTABROO	1.2
Passwd::Linux	EESTABROO	1.2
Win32API::Net	JDB	0.12
Win32::Security(::NamedObject, ::ACL)	TEVERETT	0.50
Win32::OLE	JDB	0.1709
Term::Prompt	PERSICOM	1.04
Crypt::PasswdMD5	LUISMUNOZ	1.3
DBM::Deep	RKINYON	1.0014
Readonly	ROODE	1.03

模块名	CPAN ID	版本
Expect	RGIERSIG	1.21
File::Path (随 Perl 发布)	DLAND	2.07
Win32::FileOp	JENDA	0.14.1

更多参考资料

使用一组核心数据库来自动生成配置文件，这是我在本书中始终提倡的最佳实践。而我的这个方法是从Rémy Evard那里学来的。虽然现在已经有很多人在使用这样的配置技巧，但我最初是在继承他设立的Tenwen计算环境（参考<https://www.usenix.org/publications/library/proceedings/lisa94/evard.html>上的Tenwen文献）的时候接触到这个思想的。请仔细阅读“Implemented the Hosts Database”这一节。

<http://www.rpi.edu/~finkej/>，这里有很多Jon Finke写的关于如何使用关系数据库来进行系统管理的文章。他还有些文章发布在LISA conference，请参考<http://www.usenix.org>中的文档。

Unix密码文件

<http://www.freebsd.org/cgi/man.cgi>是FreeBSD项目用来存放*BSD和其他Unix变种的在线手册的地方。在这里可以方便地比较各种操作系统在文件格式和用户管理命令（比如useradd）上的差异。

《Practical Unix & Internet Security》（Third Edition），O'Reilly出版，作者是Simson Garfinkel，这本书是关于密码文件的绝佳入门指南。

Windows用户管理

<http://Jenda.Krynicky.cz>也是一个用于用户管理的有用Win32模块的网站。

<http://aspn.activestate.com/ASPN/Mail>承担了Perl-Win32-Admin和Perl-Win32-Users两个邮件列表的运行。这两个列表（包括其归档文档）都是Windows Perl程序员不应该错过的宝库。

《Win32 Perl Programming: The Standard Extensions》（Second Edition）和《Win32 Perl Scripting: The Administrator's Handbook》这两本书都是Dave Roth所著（Sams 2001和2002出版）。虽然它们看上去有些过时，但其实仍然是Win32 Perl模块开发方面的最佳参考资料。

另外还有些Robbie Allen所著（或者合著）的书，如《Active Directory》（Third Edition）（O'Reilly）、《Active Directory Cookbook》（Second Edition）（O'Reilly）、《Managing Enterprise AD Services》（Addison-Wesley）、《Windows Server Cookbook》（O'Reilly）、《Windows Server 2003 Networking Recipes》（Apress）、《Windows Server 2003 Security Cookbook》（O'Reilly）以及Windows XP Cookbook（O'Reilly）。以上都是非常值得一读的书，但其实我要推荐的还有另外一个宝藏。Allen有个站点在<http://techtasks.com>，上面列出了书中所有例子的代码，其中有多种语言的实现（包括用Perl实现的VBScript代码）。这是在这个编程领域不可多得的资源，所以买下他的书和代码真是非常值得。

<http://win32.perl.org>有一个专用于Win32-Perl所有相关事情的维基百科。这个站点上的PPM资源库对ActiveState Perl的用户来说也很好用。

<http://learning.microsoft.com>是Microsoft资源工具包的主页。然而微软的URL规划有些奇怪，所以<http://www.microsoft.com/downloads/>才是用来下载资源工具包的网址，可以在上面搜索“resource kit”。

用户活动

前面几章，我们讨论了用户身份以及如何管理并保存账户信息。现在，让我们看看如何对正在使用系统或网络的用户进行管理。

一般来说，用户的行为可以分为以下四类：

进程操作

用户在机器上能运行进程，进程可以被创建、杀死、暂停和恢复，它们会去竞争有限的系统资源，因此可能会造成一系列资源分配的问题，这个时候往往需要系统管理员插手来进行管理。

文件操作

大多数时候，用户进程与文件系统中的文件和目录的交互包括读、写、创建、删除等，但在Unix中却并非这么简单。Unix系统将文件系统作为一个中转站，它不仅仅负责文件存储，设备控制、输入/输出，甚至一些进程控制和网络访问操作也是通过文件读写来完成的。之前在第2章中我们介绍了文件系统，而在这一章我们将从用户管理的角度继续探讨这个话题。

网络应用

在我们管理的机器上，用户能通过网络发送和接收数据。这本书的很多章节都有与网络相关的内容，但在本章，我们将从另一个角度来讨论这个话题。

特定操作系统的活动

最后剩下的是用户可以通过各种API来访问的特定操作系统的特性。包括对图形用户界面元素的控制、共享内存的使用、共享文件的编程接口，声音控制等。这个类别涉及面太广泛，要在本书中覆盖全是不可能的。我建议你在指定操作系统的网站上查找更多的信息。

进程管理

首先，我们讨论如何用Perl来处理前三类用户行为。因为我们的兴趣在于用户管理，所以重点是如何处理其他用户启动的进程。

基于Windows的操作系统进程控制

我们将简单了解一下在Windows上进程控制的四种不同方法，每种方法都能带来一些有趣的启示，相信这些小窍门能帮你解决很多其他领域的问题。我们关注的主要是两个任务：如何找到所有正在运行的进程以及如何杀掉指定的进程。

使用外部程序

有很多程序可以帮助我们罗列进程或对进程进行操作。本书的第一版介绍了Windows 2000 Resource Kit中的和

kill.exe

程序。目前，这两个程序仍然可以从微软的网站下载，而且它们在最新版本的操作系统上也都能够正常工作。另外还有一组很棒的名叫Sysinternals的实用程序集，最早由 Mark Russinovich和Bryce Cogswell发布在他们的Sysinternals网站上，现在你可以在微软的网站上找到（请查看这一章末尾的参考资料）。这个工具程序集包括了一套叫做PsTools的工具程序，这套程序可以提供微软自带的标准工具所没有的功能。

作为第一个例子，我们将使用微软操作系统自带的两个程序。这两个名为

tasklist.exe

和

taskkill.exe

的小程序能出色地完成很多任务，对编写脚本来说它们也是很好的选择（尤其是在无法下载其他程序的时候）。

默认情况下，

tasklist

的输出会是一个很宽的表格，这有时会导致阅读上的困难。加上格式化选项/F0 list后它的输出会变成这样：

```
Image Name:  System Idle Process
PID:         0
Session Name: Console
Session#:    0
Mem Usage:   16 K
Status:      Running
User Name:   NT AUTHORITY\SYSTEM
CPU Time:    1:09:06
Window Title: N/A
```

```
Image Name:  System
PID:         4
Session Name: Console
Session#:    0
Mem Usage:   212 K
Status:      Running
```

```
User Name:    NT AUTHORITY\SYSTEM
CPU Time:    0:00:44
Window Title: N/A
```

```
Image Name:   smss.exe
PID:          432
Session Name: Console
Session#:     0
Mem Usage:    372 K
Status:       Running
User Name:    NT AUTHORITY\SYSTEM
CPU Time:     0:00:00
Window Title: N/A
```

```
Image Name:   csrss.exe
PID:          488
Session Name: Console
Session#:     0
Mem Usage:    3,984 K
Status:       Running
User Name:    NT AUTHORITY\SYSTEM
CPU Time:     0:00:08
Window Title: N/A
```

```
Image Name:   winlogon.exe
PID:          512
Session Name: Console
Session#:     0
Mem Usage:    2,120 K
Status:       Running
User Name:    NT AUTHORITY\SYSTEM
CPU Time:     0:00:08
Window Title: N/A
```

另外一个tasklist的格式化选项可以使它的输出更容易用Perl解析，这个选项就是CSV(Comma/Character Separated Values，以逗号/特定字符分隔的值)。我们将在第5章中进一步讨论如何处理CSV文件，这里我们只提供一个小范例来说明如何解析上面提到的数据：

```
use Text::CSV_XS;

my $tasklist = "$ENV{'SystemRoot'}\\SYSTEM32\\TASKLIST.EXE";
my $csv = Text::CSV_XS->new();

# /v = 冗余 (verbose)，包括用户名字段；/FO CSV = 输出 CSV 格式；/NH = 不要列头
open my $TASKPIPE, '-|', "$tasklist /v /FO CSV /NH"
    or die "Can't run $tasklist: $!\n";

my @columns;
while (<$TASKPIPE) {
    next if /^$/; # 输出中跳过空行
    $csv->parse($_) or die "Could not parse this line: $_\n";
    @columns = ( $csv->fields() )[ 0, 1, 6 ]; # 取出进程名、PID 和用户名字段
```

```

    print join( ': ', @columns ), "\n";
}

close $TASKPIPE;

```

`tasklist` 还能提供一些其他有趣的信息，比如列出某个进程正在使用的动态链接库（DLL）。你可以用 `/?` 开关来查看它的详细用法。

刚才提到的另一个程序是 `taskkill.exe`，它也同样容易使用。它的参数可以是任务名（也就是所谓的“image name”），也可以是进程ID，还可以用更复杂的过滤器来指定需要杀死的进程。我建议为了安全起见，尽量使用进程ID作为参数，因为用进程名作为参数时很容易误杀同名进程。

`taskkill` 提供两种不同的方式来结束进程。第一种是比较常规的方式：`taskkill.exe /PID <process id>`，它会要求指定的进程正常结束自己。然而，如果我们在命令行加上 `/F` 参数，也就是 `taskkill.exe /F /PID <process id>`，那么它会像Perl自带的 `kill()` 函数一样强行终止指定的进程。

使用 Win32::Process::Info 模块

第二种方式^[注1]是使用 Thomas R. Wyant 的 `Win32::Process::Info` 模块。这个模块非常容易使用。首先，创建一个进程信息对象，像这样：

```

use Win32::Process::Info;
use strict;

# 运行这个脚本的用户必须拥有 DEBUG 级别的权限
my $pi = Win32::Process::Info->new( { assert_debug_priv => 1 } );

```

这个 `new()` 方法可以接受一个可选的哈希引用作为配置信息参数。在本例中我们把配置变量 `assert_debug_priv` 设为真，因为我们要求这个程序使用 debug 级别的权限去获取相关信息。如果你想取得所有进程信息，这个参数是必需的，不然你会发现这个模块没法取得某些进程的所有者信息（这是 Windows 的安全机制导致的）。这个模块的说明文档中有一些很夸张的关于这个配置项的警告信息，但我自己在使用中是没有碰到过任何问题，当然在你学习我的使用方法前，还是应该仔细阅读相关的文档。

接下来，让我们取得这个机器上的进程信息：

```

my @processinfo = $pi->GetProcInfo();

```

注1： 在本书的第一版中，本节叫做“使用 Win32::IProc 模块”，Win32::IProc 模块被换掉的原因我会在第131页“昙花一现的模块”中说明。

于是现在 @processinfo 就是一个包含多个匿名哈希引用的数组了。其中每个匿名哈希都有数个关键字（如Name、ProcessId、CreationDate和ExecutablePath）及其对应的值。要想显示这些进程信息，我们可以用和上一小节中范例同样的方法，代码如下：

```
use Win32::Process::Info;

my $pi = Win32::Process::Info->new( { assert_debug_priv => 1 } );
my @processinfo = $pi->GetProcInfo();

foreach my $process (@processinfo) {
    print join( ':',
        $process->{'Name'}, $process->{'ProcessId'},
        $process->{'Owner'} ),
        "\n";
}
```

于是我们又得到这样的输出：

```
System Idle Process:0:
System:4:
smss.exe:432:NT AUTHORITY\SYSTEM
csrss.exe:488:NT AUTHORITY\SYSTEM
winlogon.exe:512:NT AUTHORITY\SYSTEM
services.exe:556:NT AUTHORITY\SYSTEM
lsass.exe:568:NT AUTHORITY\SYSTEM
svchost.exe:736:NT AUTHORITY\SYSTEM
svchost.exe:816:NT AUTHORITY\NETWORK SERVICE
svchost.exe:884:NT AUTHORITY\SYSTEM
svchost.exe:960:NT AUTHORITY\SYSTEM
svchost.exe:1044:NT AUTHORITY\NETWORK SERVICE
svchost.exe:1104:NT AUTHORITY\LOCAL SERVICE
ccSetMgr.exe:1172:NT AUTHORITY\SYSTEM
ccEvtMgr.exe:1200:NT AUTHORITY\SYSTEM
spoolsv.exe:1324:NT AUTHORITY\SYSTEM
...
```

Win32::Process::Info 可以提供的进程信息比上面的多很多（但也许你用不到）。另外它还有一个非常有用的特性：它能显示特定进程或者所有进程的进程树。这能让你显示每个进程的子进程（也就是由某个进程衍生出来的子进程列表）以及由这些子进程再衍生出来的其他子进程，以此类推。

因此，举例来说，如果我们想显示上面的某个进程所衍生出的所有子进程，可以这样写：

```
use Win32::Process::Info;
use Data::Dumper;

my $pi = Win32::Process::Info->new( { assert_debug_priv => 1 } );
# 选 PID 884 是因为它的子进程数量相对较少，对于本例的显示来说不会过长
my %sp = $pi->Subprocesses(884);
```

```
print Dumper (\%sp);
```

这段代码会输出：

```
$VAR1 = {
    '3320' => [],
    '884' => [
        3320
    ]
};
```

结果说明，这个`svchost.exe`（PID为884）进程有一个PID为3320的子进程，并且这个子进程再也没有其他的子进程了。

使用GUI控制模块（Win32::Setupsup和Win32::GuiTest）

可以说，在我们展示的所有方法中，第三种方法是最有趣的。这一节要介绍的模块是Jens Helberg写的Win32::Setupsup，以及Ernesto Guisado、Jarek Jurasz和Dennis K. Paulsen合写的Win32::GuiTest。它们的功能类似，只是在实现上稍有不同。我们主要介绍的是Win32::Setupsup，有些特殊例子会使用Win32::GuiTest。

注意：考虑到模块的现状，这里必须说明，Win32::Setupsup模块自从2000年10月之后就没有更多的开发迹象，而且它也比较难下载（请查看这一章末尾的参考资料）。不过它仍然能正常工作，而且还有一些Win32::GuiTest不具备的功能，所以我还是介绍这个模块。如果你对它的现状感到担忧，那我建议你尝试Win32::GuiTest，看看它的功能是否足够。

Win32::Setupsup被命名为“Setupsup”是因为它的设计初衷是为了对软件安装（常常需要运行`setup.exe`程序）提供支持。

有些安装程序有所谓的“静默安装”模式，能进行无需人机交互的安装。在这种模式中，它们不会提问，也不必总是按“确定”按钮，这使管理员的工作更加轻松。而那些不能这样安装的软件则会让系统管理员头疼，应该说这样的软件实在不少。Win32::Setupsup就是为了这种让人发狂的软件安装而设计的：它能从正在运行的进程中获得必要的信息，从而驱动安装过程的自动流转。

注意：如果需要安装Win32::Setupsup，请参考第3章的“本章所用模块”一节。

有了Win32::Setupsup之后，获取正在运行的进程列表就非常容易了。范例如下：

```
use Win32::Setupsup;
use Perl6::Form;
```


结束进程也非常容易：

最后两个参数是可选的。第二个参数杀掉进程并相应地设置其退出值（如果不设置的话，退出值会是0）。第三个参数使你能杀掉系统进程，当然你得拥有Debug Programs用户权限才行。

好的，让我们结束以上乏味的介绍。下面进一步探讨如何与运行中进程打开的图形界面（也就是窗口）交互，这使得进程操纵上升到另一个高度。要列出桌面上的所有可用窗口，可以这样写：

现在 `@windowlist` 包含了窗口句柄的列表，而这个列表打印出来就是一堆数字而已。要了解窗口的详细信息，可以使用另外的函数。比如，可以用 `GetWindowText()` 函数来获取每个窗口的标题，代码如下：

用户活动 | 109

```

    }
    else {
        warn "Can't get text for $whandle"
        . Win32::Setupsup::GetLastError() . "\n";
    }
}

```

下面是一些样本输出:

```

66130: chapter04 - Microsoft Word
66184: Style
194905150:
66634: setupsup - WordPad
65716: Fuel
328754: DDE Server Window
66652:
66646:
66632: OleMainThreadWndName

```

这里你能看到某些窗口有标题, 某些没有。有经验的读者还能发现一些有趣的现象。66130号窗口属于一个正在运行的微软Word会话(没错, 这正是用来编写这一章的程序实例); 而66184号窗口看起来和这个Word会话有些关系。那么我们怎么验证这种联系呢?

Win32::Setupsup的EnumChildWindows()函数能用来查询某个窗口的子窗口。现在我们要用它来显示窗口之间的层次关系:

```

use Win32::Setupsup;

my @windowlist;
# 获取窗口列表
Win32::Setupsup::EnumWindows( \@windowlist )
    or die 'process list error: ' . Win32::Setupsup::GetLastError() . "\n";

# 把窗口列表转化成哈希
# 注意: 这个转化导致哈希中存放的是窗口句柄的数字格式, 而不是原始句柄。
# 像 GetWindowProperties 这样的函数不支持这种数字格式的参量, 稍后会看到。
# 请留意这个问题。
my %windowlist;
for (@windowlist) { $windowlist{$_}++; }

# 检查窗口的子窗口
my %children;
foreach my $whandle (@windowlist) {
    my @children;
    if ( Win32::Setupsup::EnumChildWindows( $whandle, \@children ) ) {

        # 为每个窗口保存排序过的子窗口列表
        $children{$whandle} = [ sort { $a <=> $b } @children ];

        # 从哈希中删除子窗口, 我们不必直接对它们进行遍历
    }
}

```

```

        foreach my $child (@children) {
            delete $windowlist{$child};
        }
    }
}

# 遍历窗口列表并递归地打印每个窗口句柄和其（可能的）子窗口句柄
foreach my $window ( sort { $a <=> $b } keys %windowlist ) {
    PrintFamily( $window, 0, %children );
}

# 打印一个给定窗口句柄及其所有子窗口（递归地）
sub PrintFamily {

    # 起始窗口——窗口在树中的深度
    my ( $startwindow, $level, %children ) = @_;

    # 进行合适的缩进，然后打印窗口句柄
    print( ( ' ' x $level ) . "$startwindow\n" );

    return unless ( exists $children{$startwindow} ); # 没有子窗口，可以返回

    # 否则，我们需要为每个子窗口递归
    $level++;
    foreach my $childwindow ( @{ $children{$startwindow} } ) {
        PrintFamily( $childwindow, $level, %children );
    }
}

```

最后，我们还有一个需要介绍的窗口属性函数，`GetWindowProperties()`。它基本上能用来检查所有我们还没有介绍的其他属性。比如，我们可以用`GetWindowProperties()`来查询创建窗口的进程的进程ID。这让我们能进一步使用`Win32::Process::Info`这样的模块来获取其他信息。

`Win32::Setupsup`的说明文档中列出了可以查询的属性。我们这里使用其中几个属性来写一个小程序，用于打印窗口在桌面上的位置。`GetWindowProperties()`需要三个参数：窗口句柄、包含需要查询的属性的名称的数组引用以及用来存放查询结果的哈希引用。下面的代码可以完成此任务：

```

use Win32::Setupsup;

# 把窗口ID转换成 GetWindowProperties 能处理的形式
# 注意：“U” 这个 pack 模板只有在 Perl 5.6 版本之后才能使用

my $whandle = unpack 'U', pack 'U', $ARGV[0];
my %info;
Win32::Setupsup::GetWindowProperties( $whandle, ['rect'], \%info );

print "\t" . $info{rect}{top} . "\n";
print $info{rect}{left} . ' - ' . $whandle . ' - ' . $info{rect}{right} . "\n";
print "\t" . $info{rect}{bottom} . "\n";

```

这个输出还算比较美观。这里的输出显示了66180号窗口句柄所在的顶部、左边、右边和底部的位置：

```
154
272 766180- 903
595
```

`GetWindowProperties()` 返回的数据结构比较特别，它给我们的调用返回的哈希引用中只有一个名为`rect`的键，而它的值也是一个哈希引用，其中有我们关心的数据。如果你对Perl返回的窗口属性不太确定，可以试试`windowse`工具程序，它常常能让问题迎刃而解。

现在我们看过了如何识别各种窗口属性，那么我们是否可以修改这些属性呢？你可能不止一次希望修改某个窗口的标题，如果有了这个能力，我们就可以写出那种用窗口标题作为状态指示器的程序，比如：

```
“戏法进行中……已完成32%”
```

实现这个窗口属性修改只需要一个函数调用：

```
Win32::Setupsup::SetWindowText($handle,$text);
```

我们还可以设置前面看到的`rect`属性。下面的代码能让某个窗口移动到我们指定的位置：

```
use Win32::Setupsup;

my %info;
$info{rect}{left} = 0;
$info{rect}{right} = 600;
$info{rect}{top} = 10;
$info{rect}{bottom} = 500;
my $whandle = unpack 'U', pack 'U', $ARGV[0];
Win32::Setupsup::SetWindowProperties( $whandle, \%info );
```

最后，我还要介绍最最精彩的功能：可以使用 `SendKeys()` 来向桌面上的任何窗口发送任意键盘输入。比如：

```
use Win32::Setupsup;

my $texttosend = "\\DN\\Low in the gums";
my $whandle = unpack 'U', pack 'U', $ARGV[0];
Win32::Setupsup::SendKeys( $whandle, $texttosend, 0 ,0 );
```

这会给指定窗口发送“下光标键”以及一些文本。`SendKeys()`接受的参数很简单：窗口句柄、要发送的文本、一个决定是否要激活窗口以接受键盘输入的标志以及可选的键盘

输入延迟间隔。特殊的键编码需要用反斜线转义，比如刚才看到的“下光标键”。可以查看模块的说明文档来了解可用的键代码列表。

在我们进一步介绍其他有趣的管理Windows进程的方法之前，还要再介绍一个和Win32::Setupapi有相似功能的模块，它能用来做更多有趣的事情。它的名字叫做Win32::GuiTest，也能获得关于活动窗口和发送键盘序列的信息。只不过，它还有更加特殊的功能。

下面的例子来自该模块的说明文档，不过有些修改（删除了注释和错误检查，有可能的话还是请看原始例子），可以展示它的功能：

```
use Win32::GuiTest qw(:ALL);

system("start notepad.exe");
sleep 1;

MenuSelect("F&ormat|&Font");
sleep(1);

my $fontdlg = GetForegroundWindow();

my ($combo) = FindWindowLike( $fontdlg, '', 'ComboBox', 0x470 );

for ( GetComboContents($combo) ) {
    print "$_" . "\n";
}

SendKeys("{ESC}%{F4}");
```

这段代码启动了*notepad*(记事本程序)，并通过选择适当菜单项让它打开字体设置对话框，然后读取其中的内容并且打印出来。然后，程序又做了必要的操控来关闭对话框并退出程序。最终的结果是获取了系统支持的所有等宽字体列表，如下所示：

```
'Arial'
'Arial Black'
'Comic Sans MS'
'Courier'
'Courier New'
'Estrangelo Edessa'
'Fixedsys'
'Franklin Gothic Me'
'Gautami'
'Georgia'
'Impact'
'Latha'
'Lucida Console'
'Lucida Sans Unicod'
'Mangal'
'Marlett'
'Microsoft Sans Ser
```

```
'Modern'  
'MS Sans Serif'
```

再看看下面一段代码，同样取材于该模块的说明文档，稍加修改：

```
use Win32::GuiTest qw(:ALL);  
  
system 'start notepad';  
sleep 1;  
  
my $menu = GetMenu( GetForegroundWindow() );  
menu_parse($menu);  
  
SendKeys("{ESC}%{F4}");  
  
sub menu_parse {  
    my ( $menu, $depth ) = @_;  
    $depth ||= 0;  
  
    foreach my $i ( 0 .. GetMenuItemCount($menu) - 1 ) {  
        my %h = GetMenuItemInfo( $menu, $i );  
        print '    ' x $depth;  
        print "$i  ";  
        print $h{text} if $h{type} and $h{type} eq 'string';  
        print "-----" if $h{type} and $h{type} eq 'separator';  
        print "UNKNOWN" if not $h{type};  
        print "\n";  
  
        my $submenu = GetSubMenu( $menu, $i );  
        if ($submenu) {  
            menu_parse( $submenu, $depth + 1 );  
        }  
    }  
}
```

如同前一个例子，我们还是拿*notepad*来练手。这里，我们检查前台窗口中程序所有的菜单，判断顶级菜单项的数量并遍历每个菜单（打印菜单内容并检查是否有下级菜单）。如果发现更深的子菜单，就递归调用*menu_parse()*来处理。一旦我们结束菜单遍历，就可以发送关闭*notepad*窗口的键盘序列，让程序退出。

程序输出如下：

```
0  &File  
0  &New      Ctrl+N  
1  &Open...  Ctrl+O  
2  &Save     Ctrl+S  
3  Save &As...  
4  -----  
5  Page Set&up...  
6  &Print... Ctrl+P  
7  -----  
8  E&xit
```

```

1 &Edit
  0 &Undo      Ctrl+Z
  1 -----
  2 Cu&t       Ctrl+X
  3 &Copy       Ctrl+C
  4 &Paste      Ctrl+V
  5 De&lete    Del
  6 -----
  7 &Find...    Ctrl+F
  8 Find &Next      F3
  9 &Replace...    Ctrl+H
 10 &Go To...      Ctrl+G
 11 -----
 12 Select &All    Ctrl+A
 13 Time/&Date      F5
2 F&ormat
  0 &Word Wrap
  1 &Font...
3 &View
  0 &Status Bar
4 &Help
  0 &Help Topics
  1 -----
  2 &About Notepad

```

从脚本触发已知的菜单项非常棒，不过如果能判断哪些菜单项可用则更酷一些，这样就能写出更加通用的脚本。

Win32::GuiTest还有许多没有介绍的卓越功能，如用WMGetText()函数来读取窗口内的文本，以及用SelectTabItem()选中窗口中的某个分页。请参考模块的说明文档以及范例程序目录（eg）来了解更多详细信息。

有了这两个模块的帮助，我们能在更高的级别进行进程控制。现在我们可以对应用程序（以及部分操作系统）进行远程控制了，而且这些操控不必在受控程序端实现。我们不再需要命令行支持或者特殊API，我们的系统管理脚本现在正式扩展到了图形用户界面的领域。

使用Windows管理规范（WMI）

在介绍其他操作系统之前，让我们再讨论最后一种 Windows 进程控制的方法。到目前为止，你应该已经感觉到了，前面介绍的几种方法都能在进程控制之外找到更多应用机会。聪明的读者，如果你不介意漫长的学习曲线，能耐心等待最大的回报，那么基于WMI的脚本开发应该就是你的选择了。这本书的第一版把WMI称为“Futureland”，因为在写作的时候这个技术才刚刚现身。在此后的一段时间里，由于微软的大力推进，WMI这个管理框架逐渐被操作系统和某些产品接受，目前MS SQL Server和微软

Exchange都支持它。

不幸的是，WMI属于那种很快变得很复杂的技术，对初学者来说往往缺乏吸引力。它是基于面向对象模型的，不但能用来展现数据，还能描述对象之间的关系。比如，它可以在Web服务器和相关的存储设备之间建立联系，这样当存储设备出现问题的时候，Web服务器的问题也能列在问题报告中。我们这里没有办法花更多篇幅介绍这些更为复杂的技术，只能通过一些简单的案例和代码来演示WMI最粗浅的功能。

如果你想深入了解这种技术，我建议你到<http://msdn.microsoft.com>搜索WMI相关的资料。你还可以看看分布式管理任务组（Distributed Management Task Force）的网站。不过接下来的简要介绍，应该可以作为入门知识。

WMI最初是被微软称为基于Web的企业管理（Web-Based Enterprise Managerment，简称为WBEM）的一种技术，后来在实现过程中加入了一些扩展，最终成为现在的WMI。单单从名字来看，它会让人误以为是某种需要浏览器的东西，但实际上它和万维网没有任何关系。分布式管理任务组（DMTF）联盟下的公司给它起这样的名字，是因为起初想用它创造某种可以通过浏览器来完成管理任务的东西。虽然名不符实，但是WBEM确实为管理信息存储和设备状态探测定义了一种数据模型。它还为这些信息的组织、存取及交换数据提供了一系列规范。WBEM同时也为访问由其他管理协议提供的数据库提供了一种集中统一化的前端，如简单网络管理协议（SNMP，我们将在第12章对其进行讨论）以及通用管理信息协议（CMIP）。

WBEM世界中的数据是按照通用信息模型（Common Information Model,CIM）来组织的。CIM是WBEM/WMI的功能强大和难以捉摸的根源所在。它提供了一种可扩展的数据模型，包含了你想管理的任何物理或逻辑实体的对象和类。举例来说，有表示整个网络的类，也有表示特定机器中单个插槽的对象，还有各种硬件配置对象，以及应用程序的配置对象等。在这之上，CIM还允许我们自己定义类来表示其他对象间的关系。

这种数据模型在文档中由两部分组成：CIM规范和CIM模式（Schema）。前者描述CIM如何规范数据，以及它和之前介绍的其他管理标准的映射关系等；后者定义了哪些是CIM的对象。这种划分可能会让你联想到SNMP的SMI和MIB之间的关系（见附录G和第12章）。

在实践中，一旦你掌握了数据的展现方式，以后你碰到的大部分问题就都应该是关于CIM模式，而不是CIM规范的。CIM模式的管理对象格式（Managed Object Format，MOF）是比较易读的。

CIM模式分为两层：

- 在各种类型的WBEM交互中都很有用的对象和类的核心模型（core model）。
- 和供应商、操作系统无关的常用对象的通用模型（common model）。在通用模型中，有15种特定领域，包含系统、设备、应用程序、网络及物理环境。

在这两层之上，可以构建任意数量的扩展模式（extension schema）用来为提供商和特定操作系统信息定义对象和类。WMI是WBEM的一种实现，它广泛使用了这种扩展机制。

WMI和一般WBEM实现的最大差别在于Win32模式，这是在核心模型和通用模型基础上构建的、专门针对Win32特有信息的扩展模式。WMI还将Win32特有的访问CIM数据的机制加至通用WBEM框架。^[注2]正是因为有了这种扩展模式和数据访问的机制，我们才能介绍如何在Perl中使用WMI来实现进程控制。

WMI提供了两种不同的方式来获取管理数据：面向对象的方式和基于请求的方式。前者需要指定特定的对象或包含你所需信息的对象容器；后者需要构造一个类似SQL的^[注3]查询，这个查询会返回包含你所需数据的对象的结果集。我们会针对每种方式给出一段简单的范例，让你了解它们分别是如何工作的。

下面展示的Perl代码看起来并不是特别复杂，你可能会奇怪之前我们为什么会说“它很快变得很复杂”。这些代码之所以看起来简单是因为：

- 这里我们只接触到WMI的皮毛。我们甚至都没有涉及到“关联”这样的主题（即对象和类之间的关系）。
- 我们需要做的管理操作也比较简单。这里涉及的进程控制只是查询当前运行的进程，并在需要的时候可以结束它们就好了。使用Win32模式扩展，这些操作都比较容易实现。
- 我们的范例并没有涉及如何将WMI文档中的VBScript/JScript代码转换成Perl代码。你可以参考附录F来获得与代码翻译相关的帮助。
- 我们的范例向你隐藏了繁琐的黑箱调试过程。当与WMI相关的Perl代码出错的时候（尤其是通过面向对象方式访问CIM的代码），它所提供的有意义的调试信息少之又少。你也许会得到出错信息，但这些信息从不会像ERROR: YOUR EXACT PROBLEM IS... 这样直接反映问题。一般你得到的消息会像wbemErrFailed 0x8004100这样，或者干脆就是一个空数据结构。公平地说，这种不透明性和Perl在这个过程中

注2： 尽管微软希望这套数据访问机制被广泛接受，但实际在非 Win32 环境中你很难找到它们的身影，所以我称它们为“Win32 特有的”。

注3： 微软为此还引入了一种简化的类似 SQL 的查询语言，叫做 WQL。他们还曾经提供过基于 ODBC 的数据访问方法，但是这种方法在新发布的系统中已经被废弃了。

的角色有关：它在这个比较复杂的多层操作中扮演的只是一个前端角色。当系统中有些地方出问题时，Perl自己并不能获得有价值的反馈信息。

我知道这听起来有些让人不爽，所以在我们开始接触代码之前先给出一些可能对你有用的建议：

- 阅读所有与Win32::OLE模块相关的能让你上手的代码范例。ActiveState的Win32-Users邮件列表归档，位于<http://aspn.activestate.com/ASPN/Mail>，是这类代码很好的来源。如果你把这些代码和等效的VBScript代码范例进行对比，你将会明白它们之间进行转换的窍门。另外附录F和第9章中的“活动目录服务接口”一节也可能对你有所帮助。
- 把Perl调试器当作你的好朋友，用它来测试书中的代码片段，这种练习是非常有必要的。当然CPAN上还有一些可用REPL^[注4]模块，如App::REPL、Devel::REPL和Shell::Perl，它们可以让你在“谈笑之间”完成快速原型设计。另外，某些集成开发环境（Integrated Development Environment，IDE）也能提供类似功能。
- 手边放一份WMI SDK的拷贝。这份文档及其中的VBScript代码范例十分有用。
- 经常使用WMI SDK中的WMI对象浏览器。它能帮助你了解很多相关情况。

好了，现在让我们进入本节和Perl相关的部分。我们首要的任务是弄清楚可以获取哪些Windows进程信息以及我们如何与这些信息交互。

首先需要连接WMI名称空间（namespace）。WMI SDK中描述名称空间是“负责将类和实例分组以控制其作用域和可见性的单元”。在本例中，我们感兴趣的是连接至标准的cimv2名称空间的根目录，那里包含我们感兴趣的所有数据。

另外我们还需要用合适的安全权限及模拟级别连接系统。我们的程序需要获得调试进程的权限以及“替身”的身份；换句话说，它需要以调用该脚本的用户的身份来运行。当我们建立了这个连接后，我们将取得一个Win32_Process对象（在Win32模式中定义过）。

要想创建连接并取得对象有两种方式，一种简单，一种复杂。作为第一个例子，我们将分别介绍这两种方式，从而让你了解它们各自的含义。下面是带注释的复杂版本：

```
use Win32::OLE('in');

my $server = ''; # 连接到本地机器
```

注4：REPL是Read-Eval-Print Loop的缩写，起源自LISP（LISt Processing，表处理程序）世界的一个术语。REPL能让你在提示符下输入代码，然后用相应语言的解释器执行代码并返回结果给你检查。

```
# 获得一个 SWbemLocator 对象
my $lobj = Win32::OLE->new('WbemScripting.SWbemLocator') or
    die "can't create locator object: ".Win32::OLE->LastError()."\n";

# 将模拟级别设为 "impersonate"
$lobj->{Security_}->{impersonationlevel} = 3;

# 使用它来获得 SWbemServices 对象
my $sobj = $lobj->ConnectServer($server, 'root\cimv2') or
    die "can't create server object: ".Win32::OLE->LastError()."\n";

# 获得模式对象
my $procschm = $sobj->Get('Win32_Process');
```

这种复杂方式涉及了以下问题：

- 取得定位对象，用它找到服务器连接对象。
- 设置模拟级别（impersonation level），使程序以用户自己的权限运行。
- 通过定位对象获得至cimv2 WMI名称空间的服务器连接。
- 使用这个服务器连接取得Win32_Process对象。

这种方式在你要操作中间对象的情况下会比较有用。然而，我们也可以用COM moniker的显示名称来一步完成这种操作。WMI SDK中的描述：“在通用对象模型（COM）中，*moniker*是封装位置和绑定其他COM对象的标准机制。*moniker*的文字表示称为显示名称(display name)。”下面以比较简单的方式实现了和上面代码一样的功能：

```
use Win32::OLE('in');

my $procschm = Win32::OLE->GetObject(
    'winmgmts:{impersonationLevel=impersonate}!Win32_Process')
    or die "can't create server object: ".Win32::OLE->LastError()."\n";
```

现在我们手上有一个Win32_Process对象，我们可以用它来处理（这里主要是显示）模式中定义的Windows进程。这意味着我们可以使用模式中定义的Win32_Process属性和方法。实现代码比较简单，唯一有些奇妙的是Win32::OLE in操作符的使用。为了解释这个，我们需要扯两句题外话。

我们的\$procschm对象有两个特殊的属性，分别是Properties_和Methods_。它们各自拥有一些特殊子对象，在COM术语中被称为集合对象（collection object）。集合对象其实是其他对象的父容器；在这里，集合对象保存的是模式的属性方法描述对象。in操作符会返回一个数组，其中包含指向容器对象的各个子对象的引用。^[注5]一旦取得了这个数组，我们便可以遍历它，并返回各个子对象的Name属性。代码看起来是这样：

```

use Win32::OLE('in');

# 连接到名称空间，设置模拟级别，使用显示名称来获取 Win32_process 对象
my $procschm = Win32::OLE->GetObject(
    'winmgmts:{impersonationLevel=impersonate}!Win32_Process'
    or die "can't create server object: ".Win32::OLE->LastError()."\n";

print "--- Properties ---\n";
print join("\n",map {$_->{Name}}(in $procschm->{Properties_}));
print "\n--- Methods ---\n";
print join("\n",map {$_->{Name}}(in $procschm->{Methods_}));

```

在 Windows XP SP2 的机器上输出看起来像这样：

```

--- Properties ---
Caption
CommandLine
CreationClassName
CreationDate
CSCreationClassName
CSName
Description
ExecutablePath
ExecutionState
Handle
HandleCount
InstallDate
KernelModeTime
MaximumWorkingSetSize
MinimumWorkingSetSize
Name
OSCreationClassName
OSName
OtherOperationCount
OtherTransferCount
PageFaults
PageFileUsage
ParentProcessId
PeakPageFileUsage
PeakVirtualSize
PeakWorkingSetSize
Priority
PrivatePageCount
ProcessId
QuotaNonPagedPoolUsage
QuotaPagedPoolUsage
QuotaPeakNonPagedPoolUsage
QuotaPeakPagedPoolUsage
ReadOperationCount
ReadTransferCount
SessionId

```

注5：参阅第9章的“活动目录服务接口”一节来了解 in 操作符的另一种典型用法。

```

Status
TerminationDate
ThreadCount
UserModeTime
VirtualSize
WindowsVersion
WorkingSetSize
WriteOperationCount
WriteTransferCount
--- Methods ---
Create
Terminate
GetOwner
GetOwnerSid
SetPriority
AttachDebugger

```

让我们认真处理当前的业务需求。要取得运行中进程的列表，我们需要访问所有的 Win32_Process 对象实例：

```

use Win32::OLE('in');

# 将先前的初始化代码放在循环中反复执行

my $sobj = Win32::OLE->GetObject(
    'winmgmts:{impersonationLevel=impersonate}')
or die "can't create server object: ".Win32::OLE->LastError()."\n";

foreach my $process (in $sobj->InstancesOf("Win32_Process")){
    print $process->{Name}." is pid #".$process->{ProcessId},"\n";
}

```

我们初始的显示名称没有包括指向特定对象的路径（也就是说省略了!Win32_Process）。结果我们得到了一个服务器连接对象。当我们调用InstancesOf()方法的时候，它会返回一个包含特定对象所有实例的集合对象。接下来的代码会依次访问各个对象，并打印出它们的Name和ProcessId属性。这样就输出了当前所有运行中进程的列表。

如果想在遍历各个进程时顺便搞点破坏，我们可以使用先前介绍过的一个方法来替代打印进程信息的代码：

```

foreach $process (in $sobj->InstancesOf("Win32_Process")){
    $process->Terminate(1);
}

```

上面这段代码会终止每个正在运行的进程。我不建议你运行这段代码，你可以按你的特定需求有选择性地对它进行修改。

在我们结束这一节之前，有必要提到最后一点。在本节前面的部分，我提到了用WMI获取信息有两种方式：面向对象的方式和基于查询的方式。到目前为止，我们介绍了比较直观的面向对象的方式。为了激发你的好奇心，下面介绍基于查询的方式并给出代码范例。首先，让我们从修改之前的代码范例开始。这里黑体字显示的行是关键改动部分，它使用的是WQL语句来替代InstancesOf()获取所有进程对象：

```
use Win32::OLE('in');

my $obj = Win32::OLE->GetObject('winmgmts:{impersonationLevel=impersonate}')
    or die "can't create server object: " . Win32::OLE->LastError() . "\n";

my $query = $obj->ExecQuery('SELECT Name, ProcessId FROM Win32_Process');
foreach my $process ( in $query ) {
    print $process->{Name} . ' is pid #' . $process->{ProcessId}, "\n";
}
```

现在我们可以上面黑体字显示的行中使用类似 SQL 的查询语句。举例来说，如果我们只想看到系统正在运行的 *svchost.exe* 进程的进程 ID，我们可以这样写：

```
use Win32::OLE('in');

my $obj = Win32::OLE->GetObject('winmgmts:{impersonationLevel=impersonate}')
    or die "can't create server object: " . Win32::OLE->LastError() . "\n";

my $query = $obj->ExecQuery(
    'SELECT ProcessId FROM Win32_Process WHERE Name = "svchost.exe"');
print "SvcHost processes: "
    . join( ' ', map { $_->{ProcessId} } ( in $query) ), "\n";
```

WQL可以处理其他类似SQL的查询语法。比如说，下面合法的WQL语句可以取得当前所有正在运行的名称以“svc”开头的进程的信息：

```
SELECT * from Win32_Process WHERE Name LIKE "svc%"
```

如果你熟悉SQL（哪怕所有关于它的知识都仅仅来自于本书的附录D），不妨在这里尝试一下SQL语法的WQL查询。

现在你拥有了开始使用WMI进行进程控制的相关知识。WMI还有用于操作系统的很多其他部分的Win32扩展，包括注册表和事件日志工具。

我们对Windows进程控制的探究到此为止。现在让我们把注意力转移到另外一个重要的操作系统。

Unix进程控制

Unix的进程控制也有多种选择。不过幸运的是，它们的复杂度和Windows环境中的差不

多。在Unix下谈进程控制，一般指如下三种操作：

1. 枚举机器上正在运行进程的列表
2. 改变它们的优先级或者进程组
3. 结束进程

对于后两种操作，Perl都有相应的函数：`setpriority()`、`setpgrp()`和`kill()`函数。而实现第一种操作则有比较多的选择余地。要列出正在运行的进程，我们可以通过如下方法：

- 调用外部程序，如`ps`。
- 可以试着强行解读`/dev/kmem`的内容。
- 查看`/proc`文件系统的内容（仅针对支持`/proc`虚拟文件系统的Unix版本）。
- 使用`Proc::ProcessTable`模块。

让我们分别讨论这几种方法。对于那些着急的读者来说，现在我就可以告诉你，使用`Proc::ProcessTable`是我最中意的方法。你可能想直接跳到介绍这个模块的部分，但无论如何我还是建议你先去了解下其他技术，或许将来它们会对你有所帮助。

调用外部程序

所有的Unix操作系统及其衍生系统都有一个程序叫`ps`，它可以用来列出当前运行的进程。然而，在不同的Unix变种中`ps`程序在文件系统中的位置可能不同，而且它们的命令行开关也不一定都一样。使用调用外部程序这个方法有个问题：它缺乏可移植性。

还有更烦人的问题，就是解析外部程序的输出非常困难（不同的系统不同的外部程序，其输出都可能不同）。下面这一小段是`ps`在一台古老的SunOS机器上的输出：

USER	PID	%CPU	%MEM	SZ	RSS	TT	STAT	START	TIME	COMMAND
dnb	385	0.0	0.0	268	0	p4	IW	Jul 2	0:00	/bin/zsh
dnb	24103	0.0	2.610504	1092	p3	S		Aug 10	35:49	emacs
dnb	389	0.0	2.5	3604	1044	p4	S	Jul 2	60:16	emacs
remy	15396	0.0	0.0	252	0	p9	IW	Jul 7	0:01	-zsh (zsh)
sys	393	0.0	0.0	28	0	?	IW	Jul 2	0:02	in.identd
dnb	29488	0.0	0.0	68	0	p5	IW	20:15	0:00	screen
dnb	29544	0.0	0.4	24	148	p7	R	20:39	0:00	less
dnb	5707	0.0	0.0	260	0	p6	IW	Jul 24	0:00	-zsh (zsh)
root	28766	0.0	0.0	244	0	?	IW	13:20	0:00	--:0 (xdm)

注意第三行。有两列连到一起了，这就让解析输出内容变得非常麻烦，虽然不是不可能，但着实非常让人恼火。当然在另外一些Unix变种中情况可能会好一些（如稍微新一些的Sun机器就没有这个问题），但这个确实是你不得不考虑的问题。

选择这种方法所需要写的Perl代码比较直接：用`open()`函数去运行`ps`程序，然后在`while(<FH){...}`循环中取得输出，接着用`split()`函数、`unpack()`函数或者`substr()`函数去解析输出内容。关于这点，《Perl Cookbook》中有一段相关的小技巧可以参考（作者是Tom Christiansen和Nathan Torkington，由O'Reilly出版）。

检查内核进程结构

我之所以提到这个方法，纯粹是出于完整性考虑。写代码来打开`/dev/kmem`这样的设备去访问当前运行内核的内存结构也不是不可能。通过这种访问，你可以追踪到内存中当前的进程表并直接读取它。但考虑到这么做的困难程度（你得手工分析二进制结构），同时它的可移植性也极其糟糕（哪怕相同操作系统的不同版本都有可能让你的程序出问题），所以我强烈建议你不要使用这种方法。^[注6]

假如你真的要用这个方法，请先复习一下Perl文档中的`pack()`函数、`unpack()`函数及内核头文件。你可以打开内核内存文件（通常是`/dev/kmem`文件），然后使用`read()`函数和`unpack()`函数来取得你所关心的内容。通过学习如`top` (<ftp://ftp.groupsys.com/pub/top>) 这样具有相关功能的程序的源代码，你可以获得不少帮助，其中应该有大段的C语言代码。相比之下，下面我们将要介绍的另一个方法会更好一些。

使用/proc文件系统

Unix 如今开始支持一些比较有意思的东西，其中一个就是现今大多数的衍生系统支持的`/proc`文件系统。这个奇妙的文件系统本身并不是用来做数据存储的。它提供了一个基于文件的接口来访问机器当前运行进程表。每运行一个进程，这个文件系统中就会出现一个以这个进程ID命名的“目录”。这个目录包含了一些关于当前运行的这个进程的相关信息“文件”。这些文件中的某个是可写入的，以此达到控制该进程的目的。

好消息是，这种概念非常聪明。坏消息是各个Unix提供商/开发者对这个概念的实现方式可能不同。造成的结果是，`/proc`目录下的文件往往是和各自的实现方式相关的，不同的实现方式其文件名和格式都有可能不一样。为了搞清楚哪个文件是可用的、它们包含什么内容，你不得不参考你的系统上的`procfs`或`mount_procfs`的手册页（相关信息一般在手册的第4、5或8节）。

`/proc`文件系统的可移植性相对比较好，可以用它来枚举当前运行的进程。假如我们只想列出进程ID和它们各自的属主，我们可以使用Perl的目录及`lstat()`操作符：

```
opendir my $PROC, '/proc' or die "Unable to open /proc:$!\n";
```

注6：稍后我们将介绍 `Proc::ProcessTable` 模块，你不需要自己写代码，它可以帮你做这些事情。


```
# 仅对 /proc 目录下像 PID 的项做 stat 操作
for my $process (grep /^d+$/, readdir($PROC)){
    print "$process\t". getpwuid((lstat "/proc/$process")[4])."\n";
}

closedir $PROC;
```

假如你想了解更多有关进程的信息，则需要打开并用`unpack()`解包 */proc*目录下相应的二进制文件。通常这个文件的名称会是*status*和*psinfo*。之前提到的手册页应该会提供关于这个文件中C结构的详细信息，或至少会指引你找到说明这个结构的C包含文件。由于这些是和操作系统相关的（和操作系统的版本也相关）的格式，所以你还是会遇到我们在讨论这个方法时最初说到的移植性问题。

由于到目前为止，针对我们所需支持的不同版本的不同操作系统，我们所提到的方法似乎都需要很多处理例外情况（我们希望支持的每个操作系统的每个版本各一个）的代码，这可能让你感觉有点受挫。但幸运的是还有另外一条路，能带我们走出这种困境。

使用Proc::ProcessTable模块

Daniel J. Urist（以及一些志愿者）满怀热忱地写了一个模块，叫做 `Proc::ProcessTable`，它为主要的Unix衍生系统提供了统一的访问进程表的接口。这个模块将不同*/proc*或*kmem*实现的细节差异隐藏起来，从而允许你写出移植性相对较好的代码。

仅仅加载这个模块，然后创建一个`Proc::ProcessTable::Process`对象，接着通过这个对象运行方法：

```
use Proc::ProcessTable;

my $tobj = new Proc::ProcessTable;
```

这个对象使用了Perl的绑定变量功能来展现实时的系统情况。你不需要去调用某个特定函数来刷新对象，每次你访问该对象，它都会重新读取进程表。

为了获得想要的信息，我们需要调用`table()`对象方法：

```
my $proctable = $tobj->table( );
```

`table()` 会返回一个数组引用，它里面的元素分别指向独立的进程对象。每个这样的进程对象都拥有一套函数方法可以返回关于该进程的信息。举例来说，下面的代码向我们展示了如何获取包含进程ID和属主的列表：

```
use Proc::ProcessTable;

my $tobj = new Proc::ProcessTable;
```

```
my $proctable = $tobj->table();

foreach my $process (@$proctable) {
    print $process->pid . "\t" . getpwuid( $process->uid ) . "\n";
}
```

如果想知道你所用的 Unix 衍生系统上哪些进程方法可用，你可以使用 `Proc::ProcessTable` 对象（即前面代码中的 `$tobj`）的 `fields()` 方法来返回所可用进程方法的列表。

`Proc::ProcessTable` 模块还为每个进程对象增加了三个额外的方法——`kill()` 方法、`priority()` 方法和 `pgrp()` 方法，这些对 Perl 内置函数的前端封装正好对应之前在这一节开头提到的三种操作。

让我们回过头来从全局着眼，回顾一下进程控制技术的几种用途。我们是以用户行为作为开始来研究进程控制的，所以让我们看看几个短小的着重于这些用户活动的脚本。这些例子将在 Unix 上使用 `Proc::ProcessTable` 模块，但其思想方法并不仅仅局限于这一系统平台。

第一个例子来自于 `Proc::ProcessTable` 模块的说明文档，有一些小小的修改：

```
use Proc::ProcessTable;

my $t = new Proc::ProcessTable;

foreach my $p (@{$t->table}){
    if ($p->pctmem > 95){
        $p->kill(9);
    }
}
```

如果所运行的 Unix 衍生系统支持 `pctmem()` 方法（大多数都支持），这段代码会关闭任何内存消耗达到系统内存 95% 以上的进程。就此而言，将它用在真实环境下可能太“锋芒毕露”了。所以在调用 `kill()` 命令前加上下面的代码会更合理一些：

```
print 'about to nuke ' . $p->pid . "\t" . getpwuid($p->uid) . "\n";
print 'proceed? (yes/no) ';
chomp($ans = <>);
next unless ($ans eq 'yes');
```

这里还存在一些竞争条件：可能在提示用户输入的间隙，系统的状态已经发生了改变。不过，假设我们只在处理大的进程的时候对用户进行操作提示，且这些大的进程一般不会在小段时间内改变状态，那么我们刚才的代码就没啥大问题。如果你要求更高，可以先收集需要杀掉的进程列表，然后提示用户操作，最后在真正杀掉进程前重新检查一下

这个待杀进程列表里各个进程的状态。这个方法虽然没法完全避免竞争条件，但一定程度上规避了大部分的竞争冲突。

某些情况下仅仅杀掉进程还不够。因为有时候监控某类活跃的进程也很重要，所以我们可以采取一些更加直接的措施，比如说让犯错的用户写份检查。举例来说，在我们这儿使用因特网中继聊天（Internet Relay Chat, IRC）机器人（bot）是明令禁止的。这里所说的“机器人”指的是连接到某个IRC聊天服务器并执行一些自动化操作的守护进程。尽管机器人也可以被用来做一些积极的、有建设性的工作，但目前来说大部分机器人在IRC里面并不能起到什么好的作用。而且我们也有一些安全上的考量，现在的入侵者在得手之后的第一件事（往往也是唯一的事）就是在IRC上建立某种机器人（以便自己远程控制）。因此一旦在我们的系统上出现这种东西，我们需要第一时间获知情况，相比之下杀掉这些进程倒不是最重要的。

目前最常见的机器人是*eggdrop*。如果我们想检查系统中当前是否有以这个名字运行的进程，可以使用如下的代码：

```
use Proc::ProcessTable;

my $logfile = 'eggdrops';
open my $LOG, '>>', $logfile or die "Can't open logfile for append:$!\n";

my $t = new Proc::ProcessTable;

foreach my $p ( @{ $t->table } ) {
    if ( $p->fname() =~ /eggdrop/i ) {
        print $LOG time . "\t"
            . getpwuid( $p->uid ) . "\t"
            . $p->fname() . "\n";
    }
}
close $LOG;
```

如果你想了想，说“这段代码还不够！假如有人想逃避这个检查，只要给*eggdrop*可执行文件改个名字就可以了”，没错，你是对的。我们将在本章最后一节给出更加成熟的机器人检测代码。

同时，让我们再看看另外一个Perl辅助我们进行用户进程管理的例子。到目前为止，我们所举的例子都比较反面，基本上都是着重于处理资源消耗大户和不正常的进程。所以，接下来我们看一些有积极意义的进程管理。

有时候，系统管理员需要知道系统上用户正在运行什么程序。因为有时候出于软件版权方面的考虑，我们需要知道同时运行某个程序的用户数量。这时，跟踪和管理软件许可是非常必要的。在这种情况下，一般需要有许可授权机制来控制进程数量。另外一个涉

及此需要的情况是在做机器迁移的时候，你可能需要将某个用户从一个平台迁移到另外一个平台，此时有必要确认用户在老平台上使用过的所有程序在新平台上都已经安装好了。

解决这个问题的一个方法是：将所有用户可用的非操作系统程序用包裹程序（wrapper）替换掉，这个包裹程序首先会记录正被运行的程序，然后真正运行此程序。如果需要替换的程序太多，这个方法就比较难实施。另外它也有让人觉得不快的副作用，那就是拖慢了程序调用的速度。

如果对精度的要求不是很高，只要大致检查一下哪些程序正在运行就够了，我们可以使用Proc::ProcessTable模块来解决这个问题。下面这段代码每5分钟运行一次，检查当前进程的运行情况。它对找到的进程名字保留一个计数，并且它也很聪明，不会重复计算上一次检查中已经存在的进程。每个小时它都会报告最新的检查结果，并重新开始信息采集。这个程序每间隔5分钟运行一次是因为遍历进程表是比较消耗资源的操作，我们希望这个程序尽可能不要给系统带来太多负担：

```
use Proc::ProcessTable;

my $interval = 300;    # 睡眠间隔为 5 分钟
my $partofhour = 0;    # 记录我们当前处在一个小时中的哪个位置

my $tobj = new Proc::ProcessTable;    # 创建新进程对象

my %last;              # 保存上一次运行时收集的信息
my %current;           # 保存当前运行时收集的信息
my %collection;        # 保存整个小时内运行时收集到的信息

# 永久循环，每隔 $interval 秒收集一次状态信息
# 并且每个小时将信息输出一次
while (1) {
    foreach my $process ( @{$tobj->table} ) {

        # 我们将忽略这个程序本身的进程
        next if ( $process->pid() == $$ );

        # 为下次运行保留这个进程的信息
        # （注意：这里假设你的程序的进程 PID 在每次运行时是不会被回收利用的，
        # 但在比较繁忙的系统上这个假设可能是不成立的）
        $current{ $process->pid() } = $process->fname();

        # 如果在上一次的迭代中已经有这个进程的信息，那么这次我们就忽略它
        next if ( $last{ $process->pid() } eq $process->fname() );

        # 否则，就记录下来
        $collection{ $process->fname() }++;
    }

    $partofhour += $interval;
}
```

```

%last = %current;
%current = ();
if ( $partofhour >= 3600 ) {
    print scalar localtime(time) . ( '-' x 50 ) . "\n";
    print "Name\t\tCount\n";
    print "-----\t\t-----\n";
    foreach my $name ( sort reverse_value_sort keys %collection ) {
        print "$name\t\t$collection{$name}\n";
    }
    %collection = ();
    $partofhour = 0;
}
sleep($interval);
}

# 对 %collection 中的内容按键名和键值排序 (反向的)
sub reverse_value_sort {
    return $collection{$b} <=> $collection{$a} || $a cmp $b;
}

```

这个程序还有许多可以改进的地方。比如可以针对单个用户来跟踪进程（也就是只针对某个用户运行的某个进程实例来计算），收集每天的状态情况，以漂亮的条状图来呈现信息报告等等。这取决于你想怎么做。

文件及网络操作

在本章最后一节，我们将一并介绍两种用户动作。之前我们花了不少篇幅介绍进程如何消耗CPU和内存资源，然而进程还会以用户身份操作文件系统并进行网络通信。用户管理工作也要求我们处理这些动作。

本节主题相对单一，我们的兴趣在于系统中其他用户正在进行的文件和网络操作。同时我们也只会把重点放在那些能追踪到具体用户的操作上（或追踪到某用户运行的某个进程）。有了这些目标后，让我们开始探索吧。

在Windows上跟踪文件操作

假如我们想跟踪其他用户打开的文件，最先想到的是使用第三方命令行程序*handle*，这个程序由前Sysinternals的Mark Russinovich编写。关于如何获取这个程序，请查看本章末尾的参考资料。*handle*可以给我们列出指定系统上所有被打开的文件句柄。下面摘录一段它的输出样本：

```

System pid: 4 NT AUTHORITY\SYSTEM
7C: File (-W-) C:\pagefile.sys
5DC: File (---) C:\Documents and Settings\LocalService\Local Settings\
Application Data\Microsoft\Windows\UsrClass.dat
5E0: File (---) C:\WINDOWS\system32\config\SAM.LOG

```

```

5E4: File (---) C:\Documents and Settings\LocalService\NTUSER.DAT
5E8: File (---) C:\WINDOWS\system32\config\system
5EC: File (---) C:\WINDOWS\system32\config\software.LOG
5F0: File (---) C:\WINDOWS\system32\config\software
5F8: File (---) C:\WINDOWS\system32\config\SECURITY
5FC: File (---) C:\WINDOWS\system32\config\default
600: File (---) C:\WINDOWS\system32\config\SECURITY.LOG
604: File (---) C:\WINDOWS\system32\config\default.LOG
60C: File (---) C:\WINDOWS\system32\config\SAM
610: File (---) C:\WINDOWS\system32\config\system.LOG
614: File (---) C:\Documents and Settings\NetworkService\NTUSER.DAT
8E0: File (---) C:\Documents and Settings\dNb\Local Settings\Application
Data\Microsoft\Windows\UsrClass.dat.LOG
8E4: File (---) C:\Documents and Settings\dNb\Local Settings\Application
Data\Microsoft\Windows\UsrClass.dat
8E8: File (---) C:\Documents and Settings\dNb\NTUSER.DAT.LOG
8EC: File (---) C:\Documents and Settings\dNb\NTUSER.DAT
B08: File (RW-) C:\Program Files\Symantec AntiVirus\SAVRT
B3C: File (R--) C:\System Volume Information\restore{96B84597-8A49-41EE-
8303-02D3AD2B3BA4}\RP80\change.log
B78: File (R--) C:\Program Files\Symantec AntiVirus\SAVRT\0608NAV~.TMP
-----
smss.exe pid: 436 NT AUTHORITY\SYSTEM
      8: File (RW-) C:\WINDOWS
     1C: File (RW-) C:\WINDOWS\system32

```

你也可以查询指定文件或目录的信息：

```
> handle.exe c:\WINDOWS\system32\config
```

```

Handle v3.3
Copyright (C) 1997-2007 Mark Russinovich
Sysinternals - www.sysinternals.com

```

```

System      pid: 4      5E0: C:\WINDOWS\system32\config\SAM.LOG
System      pid: 4      5E8: C:\WINDOWS\system32\config\system
System      pid: 4      5EC: C:\WINDOWS\system32\config\software.LOG
System      pid: 4      5F0: C:\WINDOWS\system32\config\software
System      pid: 4      5F8: C:\WINDOWS\system32\config\SECURITY
System      pid: 4      5FC: C:\WINDOWS\system32\config\default
System      pid: 4      600: C:\WINDOWS\system32\config\SECURITY.LOG
System      pid: 4      604: C:\WINDOWS\system32\config\default.LOG
System      pid: 4      60C: C:\WINDOWS\system32\config\SAM
System      pid: 4      610: C:\WINDOWS\system32\config\system.LOG
services.exe pid: 552   2A4: C:\WINDOWS\system32\config\AppEvent.Evt
services.exe pid: 552   2B4: C:\WINDOWS\system32\config\Internet.evt
services.exe pid: 552   2C4: C:\WINDOWS\system32\config\SecEvent.Evt
services.exe pid: 552   2D4: C:\WINDOWS\system32\config\SysEvent.Evt
svchost.exe  pid: 848   17DC: C:\WINDOWS\system32\config\systemprofile\
Application Data\Microsoft\SystemCertificates\My
ccSetMgr.exe pid: 1172  2EC: C:\WINDOWS\system32\config\systemprofile\
Application Data\Microsoft\SystemCertificates\My
ccEvtMgr.exe pid: 1200  23C: C:\WINDOWS\system32\config\systemprofile\
Application Data\Microsoft\SystemCertificates\My

```

```
Rtvsan.exe      pid: 1560    454: C:\WINDOWS\system32\config\systemprofile\
Application Data\Microsoft\SystemCertificates\My
```

通过使用-p开关给handle指定进程名，你可以获得这个进程的相关信息。

在Perl中使用这个程序很简单直接，所以我们在这里不提供任何代码范例。作为替代，让我们看看相关的更有意思的操作：审计。

Windows允许我们高效地监控文件、目录或目录树的变化。你可能会想到对目标对象重复使用stat()方法，但这种方法很耗CPU。在Windows下，我们可以让操作系统来帮我们进行监控。

有个特殊的Perl模块可以让这个工作相对不那么烦人：Christopher J. Madsen的Win32::ChangeNotify模块。还有一个由Renee Baecker写的相关的辅助模块：Win32::FileNotify模块。

昙花一现的模块

在本书的第一版，这一节描述了如何使用Amine Moulay Ramdane的Win32::AdvNotify模块进行文件系统审计。这个模块很棒，它是这个作者开发的多个优秀的Windows模块之一，能实现Win32::ChangeNotify所有的功能，并且还提供更多的功能。

但不幸的是，Ramdane对这个模块的分发条款的严苛态度让人费解。除了他自己的网站，他不允许其他任何网站提供这个模块，他也不允许自己的站点被镜像。模块的源码从未发布过。

据Wayback Machine (<http://www.archive.org/web/web.php>) 上的数据显示，到2002年的4月份，他的网站中所有的实践案例及他的优秀模块都已经消失了。于是在这不久之后，我就开始收到本书第一版读者的来信，都是关于书中使用了Ramdane模块的那些代码范例的。但此时我唯一能做的就是向他们推荐一些其他可用作替代的模块或方式。尽管如果你费劲去网上找还是能找到Ramdane的模块，但我还是把本节中相关的演示代码删掉了。因为这些模块缺乏开发者的支持（也没有其他人提供支持），就这点来说，使用它们会有风险。唉。

Win32::ChangeNotify很容易使用，但它也有一个缺点。这个模块使用Win32 API通过操作系统来获知某个目录中是否有什么发生了改变。你甚至可以取得指定类型的变化信息（如写入时间、文件或目录名称/大小等）。但问题是，如果你使用它来监视某个目录的变化，它只能告诉你这个改变发生在什么时候，却无法告诉你到底发生了什么改变。要

想知道具体的改变，就需要程序作者编写额外的独立代码来实现。这里我们不得不提到 Win32::FileNotify 模块。如果你只需要监控单个文件，那么当操作系统报告有文件发生了改变的时候，Win32::FileNotify 模块就能更进一步帮你确认被你审计的文件是否发生了改变。

因为代码量都不多，所以这两个模块的例子我们都来介绍一下。范例的目的是监视某个文件是否发生了改变：

```
use Win32::FileNotify;

my $file = 'c:\windows\temp\importantfile';

my $fnot = Win32::FileNotify->new($file);

$fnot->wait();    # 在这里，程序阻塞，直到 $file 文件发生改变

... # 文件改变后进行想要的操作
```

另外，这里还有段代码是查看目录中的改变的（如文件被创建、文件被删除或移走）：

```
use Win32::ChangeNotify;

my $dir = 'c:\importantdir';

# 监控该目录（第二个参数的意思是不要监控该目录下的子目录）
# 下所有找到的文件
my $cnot = Win32::ChangeNotify->new( $dir, 0, 'FILE_NAME' );

while (1) {

    # 阻塞 10 秒（10 000 毫秒）或直到发生改变
    my $waitresult = $cnot->wait(10000);

    if ( $waitresult == 1 ) {

        ... # 调用或引入其他代码来搞清楚到底发生了什么改变

        # 重置 ChangeNotification 对象以便于我们继续监控
        $cnot->reset;
    }
    elsif ( $waitresult == 0 ) {
        print "no changes to $dir in the last 10 seconds\n";
    }
    elsif ( $waitresult == ?1 ) {
        print "something went blooey in the monitoring\n";
        last;
    }
}
```

在Windows上跟踪网络操作

以上介绍了文件系统的监控。那关于网络访问的监控呢？在Windows下有两种相对容易

的方法来跟踪网络操作。理想情况下，作为一名管理员，你肯定希望知道哪个进程（以及相关用户）打开了网络端口。我还不知道有哪个Perl模块可以完成这个任务，但我知道有两个命令行工具的输出信息可以被Perl程序加以利用，从而实现同样的功能。第一个是netstat命令，它是系统自带的命令，但很少有人知道它可以实现这样的功能（我也有很长一段时间不知道）。下面是一部分输出样本：

```
> netstat -ano
```

Active Connections

Proto	Local Address	Foreign Address	State	PID
TCP	0.0.0.0:135	0.0.0.0:0	LISTENING	932
TCP	0.0.0.0:445	0.0.0.0:0	LISTENING	4
TCP	127.0.0.1:1028	0.0.0.0:0	LISTENING	1216
TCP	192.168.16.129:139	0.0.0.0:0	LISTENING	4
UDP	0.0.0.0:445	*:*		4
UDP	0.0.0.0:500	*:*		680
UDP	0.0.0.0:1036	*:*		1068
UDP	0.0.0.0:1263	*:*		1068
UDP	0.0.0.0:4500	*:*		680
UDP	127.0.0.1:123	*:*		1024
UDP	127.0.0.1:1900	*:*		1108
UDP	192.168.16.129:123	*:*		1024
UDP	192.168.16.129:137	*:*		4
UDP	192.168.16.129:138	*:*		4
UDP	192.168.16.129:1900	*:*		1108

第二个工具是由前Sysinternals的Mark Russinovich开发的TcpView（或者更精确地说，是这个工具包中的tcpvcon工具程序）。它有个很棒的功能，就是它能输出CSV形式的信息，如：

```
> tcpvcon -anc
```

```
TCPView v2.51 - TCP/UDP endpoint viewer  
Copyright (C) 1998-2007 Mark Russinovich  
Sysinternals - www.sysinternals.com
```

```
TCP,alg.exe,1216,LISTENING,127.0.0.1:1028,0.0.0.0:0  
TCP,System,4,LISTENING,0.0.0.0:445,0.0.0.0:0  
TCP,svchost.exe,932,LISTENING,0.0.0.0:135,0.0.0.0:0  
TCP,System,4,LISTENING,192.168.16.129:139,0.0.0.0:0  
UDP,svchost.exe,1024,*,192.168.16.129:123,*:*  
UDP,lsass.exe,680,*,0.0.0.0:500,*:*  
UDP,svchost.exe,1068,*,0.0.0.0:1036,*:*  
UDP,svchost.exe,1108,*,192.168.16.129:1900,*:*  
UDP,svchost.exe,1024,*,127.0.0.1:123,*:*  
UDP,System,4,*,192.168.16.129:137,*:*  
UDP,svchost.exe,1108,*,127.0.0.1:1900,*:*  
UDP,lsass.exe,680,*,0.0.0.0:4500,*:*  
UDP,System,4,*,192.168.16.129:138,*:*  
UDP,svchost.exe,1068,*,0.0.0.0:1263,*:*
```

```
UDP,System,4,*,0.0.0.0:445,*,*
```

使用如`Text::CSV::Simple`或`Text::CSV_XS`这样的模块来解析就是轻而易举的事了。

接下来让我们看看在Unix世界中该如何完成相同的任务。

在Unix上跟踪文件和网络操作

要在Unix上处理文件和网络操作，我们只需要使用一个方法。^[注7]我认为调用单独的第三程序是最好的方法，这是本书为数不多的会如此评价的地方之一。Vic Abell给系统管理员送了一份令人惊叹的礼物，他编写了名为*lsof* (LiSt Open Files，列出打开的文件)的程序，你可以在<ftp://vic.cc.purdue.edu/pub/tools/unix/lsof>找到它。*lsof*可以详细地显示Unix机器上当前所有打开的文件及网络连接。这个程序最令人惊叹的地方在于它强大的可移植性。它的最新版（在本书写作时）至少可以在9种Unix系统上运行（之前的版本甚至支持更多种Unix），而且支持这些Unix系统各自的多个不同版本。

下面是一段*lsof*的输出，摘录了一段我正在运行的一个进程的相关信息。*lsof*倾向于输出很长的行，所以在每行之间我插入了一个空白行以便区分：

```
COMMAND    PID USER   FD   TYPE    DEVICE  SIZE/OFF      NODE NAME
firefox-b  27189 dnb     cwd   VDIR    318,16168 36864 25760428 /home/dnb

firefox-b  27189 dnb     txt   VREG    318,37181 177864 6320643
/net/csw (fileserver:/vol/systems/csw)

firefox-b  27189 dnb     txt   VREG          136,0 56874 3680
/usr/openwin/lib/X11/fonts/Type1/outline/Helvetica-Bold.pfa

firefox-b  27189 dnb     txt   VREG    318,37181 16524 563516
/net/csw (fileserver:/vol/systems/csw)

firefox-b  27189 dnb      0u    unix      105,43      0t0 3352
/devices/pseudo/tl@0:ticots->(socketpair: 0x1409) (0x300034a1010)

firefox-b  27189 dnb      2u    unix      105,45      0t0 3352
/devices/pseudo/tl@0:ticots->(socketpair: 0x140b) (0x300034a01d0)

firefox-b  27189 dnb      4u    IPv6 0x3000349cde0 0t2121076
TCP localhost:32887->localhost:6010 (ESTABLISHED)

firefox-b  27189 dnb      6u    FIFO 0x30003726ee8      0t0 2105883
```

注7： 这个方法的可移植性最强。不同的操作系统都有它们自己相关的机制（如 *inotify*、*dnotify* 等），另外 *DTrace* 框架也很酷。Mac OS X 10.5 以上的版本拥有和我们之前看到的 Windows 上类似的审计工具（`Mac::FSEvents` 模块能让我们能很容易地访问）。然而这些可选的方法都没有我们接下来要介绍的可移植性强。

```
(fifofs) ->0x30003726de0

firefox-b 27189 dnb 24r VREG 318,37181 332618
85700 /net/csw (fileserver:/vol/systems/csw)

firefox-b 27189 dnb 29u unix 105,46 0t1742
3352 /devices/pseudo/tl@0:ticots->/var/tmp/orbit-dnb/linc
-6a37-0-47776fee636a2 (0x30003cc1900->0x300045731f8)

firefox-b 27189 dnb 31u unix 105,50 0t0
3352 /devices/pseudo/tl@0:ticots->/var/tmp/orbit-dnb/linc
-6a35-0-47772fb086240 (0x300034a13a0)

firefox-b 27189 dnb 43u IPv4 0x30742eb79b0 0t42210
TCP desktop.example.edu:32897->images.slashdot.org:www (ESTABLISHED)
```

这段输出展示了这个命令的一些强大功能。它显示了这个进程打开的当前工作目录（即 VDIR）、常规文件（VREG）、管道（FIFO）以及网络连接（IPv4/IPv6）。

在Perl中利用`lsyf`的最简单的方法就是用它特殊的“字段（field）”模式（即使用-F开关）。在这个模式中，原本类似刚才`ps`那样列格式的输出生会被截断成用特殊标签标记和分隔的字段，这让解析输出变得很简单。

这个字段模式输出有一个技巧。它会按作者称为“进程集”和“文件集”的方式来进行组织。一个进程集指的是一系列关于某个进程的字段项集合，把这里的“进程”换成“文件”就是文件集。如果我们在打开字段模式的同时使用`o`选项，那么一切就变得更清楚了，此时字段间会以NUL字符（ASCII码为0）来分隔。下面是和前面的输出内容类似的一组输出行，但这次是字段模式（NUL显示为^@）。为了方便阅读，我还是加上了空行分隔：

```
p27189^@g27155^@R27183^@cfirefox-bin^@u6070^@Ldnb^@
fcwd^@a ^@l

^@tVDIR^@Nox30001b7b1d8^@D0x13e00003f28^@s36864^@i25760428^@k90^@n/home/dnb^@
ftxt^@a ^@l

^@tVREG^@Nox3000224a0f0^@D0x13e0000913d^@s177864^@i6320
643^@k1^@n/net/csw (fileserver:/vol/systems/csw)^@
ftxt^@a ^@l

^@tVREG^@Nox30001714950^@D0x8800000000^@s35064^@i2800^@k1^@n/usr/lib/nss_files.so.1

^@tVREG^@Nox300036226c0^@D0x8800000000^@s56874^@i3680^@k1^@n/usr/
openwin/lib/X11/fonts/Type1/outline/Helvetica-Bold.pfa^@
ftxt^@a ^@l

^@tunix^@Fox3000328c550^@C6^@G0x3;0x0^@Nox300034a1010^@D0x8800
000000^@o0t0^@i3352^@n/devices/pseudo/tl@0:ticots->(socketpair:
0x1409) (0x300034a1010)^@
```

```

f1^@au^@l

^@tDOOR^@Fox3000328cf98^@C1^@G0x2001;0x1^@N0x3000178b300^@D0x13
c00000000^@o0t0^@i54^@k27^@n/var/run (swap) (door to nsdc[240])^@
f4^@au^@l

^@tIPv6^@Fox300037258f0^@C1^@G0x83;0x1^@N0x300034ace50^@d0x3000349
cde0^@o0t3919884^@PTCP^@nlocalhost:32887->localhost:6010^@TST=
ESTABLISHED^@TQR=0^@TQS=8191^@TWR=49152^@TWW=13264^@
f5^@au^@l

^@tFIF0^@Fox30003724f50^@C1^@G0x3;0x0^@N0x30003726de0^@d0x30003726
de0^@o0t0^@i2105883^@n(fifofs) ->0x30003726ee8^@
f6^@au^@l

^@tFIF0^@Fox30003725420^@C1^@G0x3;0x0^@N0x30003726ee8^@d0x30003726
ee8^@o0t0^@i2105883^@n(fifofs) ->0x30003726de0^@
f7^@aw^@lW^@tVREG^@Fox30003724c40^@C1^@G0x302;0x0^@N0x30001eadbf8^
@D0x13e00003f28^@s0^@i1539532^@k1^@n/home/dnb (fileserver:/vol/homedirs/systems/dnb)^@
f8^@au^@l

^@tIPv4^@Fox30003724ce8^@C1^@G0x83;0x0^@N0x300034ac010^@d0x
300040604f0^@o0t4094^@PTCP^@ndesktop.example.edu:32931->web
-vip.srv.jobthread.com:www^@TST=CLOSE_WAIT^@TQR=0^@TQS=0^@TWR=49640^@TWW=6960^@
f44^@au^@l

^@tVREG^@Fox3000328c5c0^@C1^@G0x2103;0x0^@N0x300051cd3f8^@
D0x13e00003f28^@s276^@i16547341^@k1^@n/home/dnb (fileserver:/vol/
homedirs/systems/dnb)^@
f45^@au^@l

^@tVREG^@Fox30003725f80^@C1^@G0x3;0x0^@N0x300026ad920^@D0x
13e00003f28^@s8468^@i21298675^@k1^@n/home/dnb (fileserver:/vol/homedirs/systems/dnb)^@
f46^@au^@l

^@tIPv4^@Fox30003724a10^@C1^@G0x83;0x0^@N0x309ab62b578^@d0x30742
eb76b0^@o0t20726^@PTCP^@ndesktop.example.edu:32934->216.66.26.
161:www^@TST=ESTABLISHED^@TQR=0^@TQS=0^@TWR=49640^@TWW=6432^@
f47^@au^@l

^@tVREG^@Fox3000328c080^@C1^@G0x2103;0x0^@N0x30002186098^@D0x
13e00003f28^@s66560^@i16547342^@k1^@n/home/dnb (fileserver:/vol/
homedirs/systems/dnb)^@
f48^@au^@l

```

让我们对这个输出进行解析。第一行是进程集（我们可以从它开头的字母p看出来）：

```

p27189^@g27155^@R27183^@cfirefox-bin^@u6070^@Ldnb^@
fcwd^@a ^@l

```

每个字段都以一个表明该字段内容的字母开头（p代表pid，c代表command，u代表uid，L代表login），并且以分隔符结束。这一行所有的字段共同组成了一个进程集。

所有它下面紧跟的行，直到另外一个进程集为止，描述的是当前这个进程集所打开的文件/网络连接。

让我们实际使用一下这个模式。如果我们想显示系统上所有打开的文件以及正在使用这些文件的PID，可以用如下代码：^[注8]

```
use Text::Wrap;

my $lsofexec = '/usr/local/bin/lsof'; # lsof 可执行文件的位置

# (F) 字段模式, NUL (0) 为分隔符, 显示 (L) 登录名, 文件(t)类型及文件(n)名称
my $lsofflag = '-FLOtn';

open my $LSOFPIPE, '-|', "$lsofexec $lsofflag"
    or die "Unable to start $lsofexec: $!\n";

my $pid;                # lsof 返回的进程 ID
my $pathname;           # lsof 返回的路径名
my $login;              # lsof 返回的登录名
my $type;               # lsof 返回的打开文件的类型
my %seen;               # 用来缓存路径名
my %paths;              # 在我们的运行过程中收集路径

while ( my $lsof = <$LSOFPIPE> ) {

    # 处理进程集
    if ( substr( $lsof, 0, 1 ) eq 'p' ) {
        ( $pid, $login ) = split( /\0/, $lsof );
        $pid = substr( $pid, 1, length($pid) );
    }

    # 处理文件集, 注意: 我们只关心“常规”文件 (与 Solaris 和 Linux
    # 上的情况类似, lsof 在不同的系统上可能会用不同的标签来标记文件和目录)
    if ( substr( $lsof, 0, 5 ) eq 'tVREG' or # Solaris
        substr( $lsof, 0, 4 ) eq 'tREG' ) { # Linux
        ( $type, $pathname ) = split( /\0/, $lsof );

        # 一个进程可能会打开同一个路径两次;
        # 下面的两行确保我们不会重复记录该路径
        next if ( $seen{$pathname} eq $pid );
        $seen{$pathname} = $pid;

        $pathname = substr( $pathname, 1, length($pathname) );
        push( @{$paths{$pathname}}, $pid );
    }
}

close $LSOFPIPE;
```

注8: 如果你不想手动解析 *lsof* 的字段模式输出, Marc Beyer 的 `Unix::Lsof` 可以帮你完成这项工作。

```
foreach my $path ( sort keys %paths ) {
    print "$path:\n";
    print wrap( "\t", "\t", join( " ", @{$paths{$path}} ) ), "\n";
}
```

这段代码只安排*lsdf*显示众多字段中的一部分。我们遍历了它的输出，从一个包含列表的哈希中收集文件名和PID。当我们读完所有输出内容后，以漂亮的格式化过的PID列表形式将文件名打印出来（感谢David Muir Sharnoff的Text::Wrap模块）：

```
/home/dnb (fileserver:/vol/homedirs/systems/dnb):
12777 12933 27293 28223
/usr/lib/ld.so.1:
10613 12777 12933 27217 27219 27293 28147 28149 28223 28352 28353
28361
/usr/lib/libaio.so.1:
27217 28147 28352 28353 28361
/usr/lib/libc.so.1:
10613 12777 12933 27217 27219 27293 28147 28149 28223 28352 28353
28361
/usr/lib/libmd5.so.1:
10613 27217 28147 28352 28353 28361
/usr/lib/libmp.so.2:
10613 27217 27219 28147 28149 28352 28353 28361
/usr/lib/libnsl.so.1:
10613 27217 27219 28147 28149 28352 28353 28361
/usr/lib/libsocket.so.1:
10613 27217 27219 28147 28149 28352 28353 28361
/usr/lib/sparcv9/libnsl.so.1:
28362 28365
/usr/lib/sparcv9/libsocket.so.1:
28362 28365
/usr/platform/sun4u-us3/lib/libc_psr.so.1:
10613 12777 12933 27217 27219 27293 28147 28149 28223 28352 28353
28361
/usr/platform/sun4u-us3/lib/sparcv9/libc_psr.so.1:
28362 28365
...
```

在我们介绍跟踪Unix文件和网络操作的最后一个范例前，先让我们回到更早些时候的那个范例，就是我们试图找到系统上运行的IRC机器人的那个。相对于查看进程表的方式，我们还有更可靠的方法来找出像机器人这样的网络守护进程。用户也许可以通过给可执行文件重命名来隐藏某个机器人，但要隐藏打开的网络连接会比较困难。一般情况下，IRC机器人会连接某个服务器的6660—7000号TCP端口。通过*lsdf*可以很容易找出满足这个条件的进程：

```
my $lsdfexec = '/usr/local/bin/lsdf';      # lsdf 可执行文件的位置
my $lsdfflag = '-FLOC -iTCP:6660-7000';    # 指定端口和其他 lsdf 标志

# 下面是一个哈希切片，用来预加载一个哈希表，其中键的存在性我们将稍后
```

```

# 再进行检查。通常会这么写：
#   %approvedclients = ('ircII' => undef, 'xirc' => undef, ...);
#   (但下面这个很酷的写法来自 Mark-Jason Dominus)
my %approvedclients;
@approvedclients{ 'ircII', 'xirc', 'pirc' } = ();

open my $LSOFPIPE, "$lssofexec $lsofflag|"
    or die "Unable to start $lssofexec:!\n";

my $pid;
my $command;
my $login;
while ( my $lsof = <$LSOFPIPE> ) {
    ( $pid, $command, $login ) =
        $lsof =~ /p(\d+)\000
                c(.+)\000
                L(\w+)\000/x;
    warn "$login using an unapproved client called $command (pid $pid)!\n"
        unless ( exists $approvedclients{$command} );
}

close $LSOFPIPE;

```

这是我们能做的最简单的检查。它会把将*eggdrop*改名为*pine*或者*tcsh*的用户抓出来，同时那些不打算隐藏其机器人的用户也一样会被抓出来。然而，它和我们之前提过的另一种方法存在相同的缺陷。如果用户足够聪明，他们可能会将机器人重命名为我们“允许的客户端”列表中的名字，为了进一步打击他们，我们至少还得再进两步：

- 使用*lsof*检查被可执行程序打开的文件的确就是我们期望它打开的文件，不会是某个用户系统中随机的二进制文件。
- 用我们的进程控制方法来检查用户运行的这个进程的确是运行在某个已经存在的shell上。如果这是某个用户唯一运行的进程（也就是说，如果用户已经注销，但仍留下这个进程在运行），那它很可能是个守护进程，即某种机器人。

这个猫捉老鼠的游戏帮我们结束了这一章。在第3章中我们提到过用户基本上是不可预测的。他们会做系统管理员事先不曾料到的事情。有句老话说：“愚蠢是不可避免的，因为愚蠢太富有创造力了。”在你使用Perl编程进行用户管理的时候，记住这个事实是非常重要的。它不但能让你写出更多健壮的程序，也能在用户做了某些意料之外的事导致你的程序出问题，让你冷静坐下来分析原因并惊叹于他们的创造性。

本章所用模块

模块名	CPAN ID	版本
Text::CSV_XS	HMBRAND	0.32
Win32::Process::Info	WYANT	1.011
Win32::Setupsup	JHELBURG	1.0.1.0
Win32::GuiTest	KARASIC	1.54
Win32::OLE (随 ActiveState Perl 发布)	JDB	0.1703
Proc::ProcessTable	DURIST	0.41
Data::Dumper (随 Perl 发布)	GSAR	2.121
Win32::ChangeNotify	JDB	1.05
Win32::FileNotify	RENEEB	0.1
Text::Wrap (随 Perl 发布)	MUIR	2006.1117

安装 Win32::Setupsup

如果你想安装Win32::Setupsup模块，它并不在ActiveState Perl默认的PPM仓库里，你需要从别的PPM仓库中安装它。在写本书时，这个很方便的第三方仓库是由加拿大温尼伯大学的Randy Kobes维护的。我强烈建议你添加该第三方仓库到PPM仓库列表中，哪怕你并不打算用Win32::Setupsup模块。添加这个仓库最方便的方法就是在命令行上输入下面这条命令：

```
$ ppm repo add uwinnipeg http://theoryx5.uwinnipeg.ca/ppms/
```

或者，如果你用的是 Perl 5.10，那么运行：

```
$ ppm repo add uwinnipeg http://cpan.uwinnipeg.ca/PPMPackages/10xx/
```

如果你使用PPM4的图形用户界面版本，则可以从“Edit”菜单中选择“Preferences”选项，然后在“Repositories”标签页中添加以上相应的PPM仓库地址。关于这个第三方PPM仓库的更多信息，请参见<http://theoryx5.uwinnipeg.ca/ppms/>。

更多参考资料

<http://aspn.activestate.com/ASPN/Mail/>上有Perl-Win32-Admin和Perl-Win32-Users两个邮件列表。这两个邮件列表及其邮件归档都是每个Win32程序员不可多得的宝贵资料。

<http://www.microsoft.com/whdc/system/pnppwr/wmi/default.msp>是现在WMI在Microsoft.com上面的主页。这个网址自第一版发布以来已经变过几次，所以如果你要找微软WMI主页的URL，还是在网上搜索一下“WMI”比较靠谱。

<http://technet.microsoft.com/sysinternals/>是（在我写这本书的时候还是）*handle*这个程序以及其他一些Windows上有用的工具程序的主页（Microsoft收购了Sysinternals并招募了其负责人，然后获得了这些工具程序）。在写本书时，<http://sysinternals.com>这个域名仍然可用，并且它会将你带到相应的微软网页去，所以如果你在微软的网站上找不到这些工具程序，可以试试这个域名。

<http://www.dmtf.org>是分布式管理小组（DMTF）的主页，同时它也是一个很好的 WBEM 信息源。

如果你还没有安装Microsoft Scriptomatic工具（写这本书时的版本号为2），你需要从<http://www.microsoft.com/technet/scriptcenter/tools/scripto2.msp>下载。这个来自“the Microsoft Scripting Guys”的Windows工具能让你在机器上使用WMI名称空间时更加得心应手。当你找到某些有意思的可以用到的东西的时候，这个工具可以帮你编写脚本来使用它。更强的是，它能帮你生成VBScript、JScript、Perl或者是Python的脚本。在本章和其他提到WMI的章节，我都强烈推崇这个工具。如果想在Vista下使用这个工具，请务必参阅第1章中有关Vista的部分。

TCP/IP名称和配置服务

这一代计算机主要使用的传输协议是运行在*Internet Protocol*（互联网协议，IP）之上的*Transmission Control Protocol*（传输控制协议，TCP）^[注1]。这两个协议的缩写常常被连在一起成为*TCP/IP*。每个加入TCP/IP网络的机器都必须至少有一个唯一的数字标识符，称为IP地址（IP address）。IP地址常常用*N.N.N.N*这样的方式来表示，如192.168.1.9。

虽然机器之间使用这样的点分地址格式进行通信很方便，但大多数人应该不会喜欢这样的格式。如果每个用户都必须记住12位的机器地址的话，TCP/IP应该不会像今天这样普及。所以必须得有一种人类能轻松记住的名字取代它，并将这个名字映射到对应地址。同时还得让机器能够自动获取自己的TCP/IP配置（即IP地址），避免人类手动配置。

这一章描述了网络名称服务的发展，正是这个创新使得我们可以使用*www.oog.org*这样的网址来访问站点，从而避免使用数字形式的IP地址。我们还会分析最典型的基于中心服务器的TCP/IP自动配置方案。当然在分析过程中我们也会不时给出如何通过Perl提高网络系统的可管理性的建议。

Host文件

第一个解决机器名到IP地址的映射的方案非常简单：创建一个标准文件来存放一组IP地址和相应的机器名。这个文件在Unix和OS X系统上是*/etc/hosts*，而在基于Windows的机器上则是*%SystemRoot%\System32\Drivers\Etc\hosts*。下面就是Unix式host文件的样子：

```
127.0.0.1    localhost
192.168.1.1  everest.oog.org    everest
192.168.1.2  rivendell.oog.org  rivendell
```

注1： 这一章主要讨论IPv4，这是目前的部署标准。IPv6（下一代IP标准）可能会在不久的将来代替它。

这个方式的局限性也非常容易发现。如果oog.org站点的网络管理员已经配置好了两台机器能彼此识别名字，而现在又有了第三台机器，那么就必须在每台机器上逐一更新配置。如果现在我们又买了第四台机器，那么就必须维护四个文件（每台机器一个）。

尽管看上去难以置信，但这确实就是Internet/ARPAnet早期的状况。在某个新站点被连接以后，所有其他想要和这个站点互访的其他站点都必须修改自己的host文件。中心host库则被命名为网络信息中心（Network Information Center, NIC），或者更精确地说是SRI-NIC，因为当时维护（定时修正并发布HOSTS.TXT文件）它的是Stanford Research Institute。为了保持信息及时更新，系统管理员需要定时从SRI-NIC的NETINFO目录匿名下载这个文件。

host文件如今还在使用中，尽管它看上去问题多多，而且有众多的替代方案（如同我们后面要看到的）。在一个小型网络环境中，如果能保证host文件及时更新并包含了网络中所有的主机，那么它就会非常有用。而且这个文件并不需要发布到网络中的每台主机，我们后面会介绍发布主机名信息的方法。哪怕这个文件只是用来供管理员不时查询，也会非常有用。

奇怪的是，host文件近几年又东山再起了。它提供了一种覆盖其他网络主机名的机制，在需要禁止对某个主机的访问的时候很有用。如果想要屏蔽对某个图片广告站点或者网络浏览习惯跟踪站点的连接，你可以把它们的主机名放在host文件中，使它映射到假的IP地址。不幸的是，如今病毒编写者也开始使用这个文件来破坏反病毒程序的自动更新。

Host文件？西部牛仔！

现在像域名服务（Domain Name Service, DNS）和动态主机配置协议（Dynamic Host Configuration Protocol, DHCP）这样的服务已经随处可见，那为什么还要和看上去有点过时的host文件打交道呢？

这是因为host文件足够简单，其语法和语义都简单到任何人能够理解。相比我们将要介绍的那些服务来说，这个优势还是比较明显的。所以如果想要避免陷入某个具体服务的实现细节或配置文件语法，我们就应该考虑它。

我们将要展示的技术可以应用至任何支持纯文本配置文件的网络服务。我们首先在host文件上使用它们，因为这是最简单的应用案例，将来你会发现更多的可以使用它们的场合。在这一章的后面我们就会在其他服务上展示其他应用案例。

如果你每次看到host文件就联想到“老牛仔”，那么请不要太当真，只是大致了解也可以。^[注2]

Perl和host文件是最佳组合，这主要是Perl在处理文本文件上面的特殊偏好导致的。我们会使用简单的host文件作为敲门砖，从而展开一系列的主题。

首先，让我们先看看host文件的分析。这其实非常简单：

```
open( my $HOSTS, '<', '/etc/hosts' ) or die "Unable to open host file:$!\n";
my %addrs;
my %names;
while ( defined( $_ = <$HOSTS> ) ) {
    next if /^#/;      # 跳过注释行
    next if /\s*$/;    # 跳过空行
    s/\s*#.*$//;      # 去除行末注释和空白符
    chomp;
    my ( $ip, @names ) = split;
    die "The IP address $ip already seen!\n" if ( exists $addrs{$ip} );
    $addrs{$ip} = [@names];
    for (@names) {
        die "The host name $_ already seen!\n" if ( exists $names{lc $_} );
        $names{lc $_} = $ip;
    }
}
close $HOSTS;
```

以上代码能遍历/etc/hosts文件，跳过那些空行和注释行，并在两个数据结构中采集将来要用到的信息。第一个数据结构是一个列表的哈希，它的键是IP地址。看起来结构如下：

```
$addrs{'127.0.0.1'} = ['localhost'];
$addrs{'192.168.1.2'} = ['rivendell.oog.org','rivendell'];
$addrs{'192.168.1.1'} = ['everest.oog.org','everest'];
```

第二个数据结构是一个单纯的哈希表，以主机名为键。对于同一个文件的分析结果，应该产生这样的%names哈希：

```
$names{'localhost'} = '127.0.0.1'
$names{'everest'} = '192.168.1.1'
$names{'everest.oog.org'} = '192.168.1.1'
$names{'rivendell'} = '192.168.1.2'
$names{'rivendell.oog.org'} = '192.168.1.2'
```

注意，在如此简单的文件分析过程中，我们也会捕获一些重要的信息。代码会检查重

注2： 那些真正的“老牛仔”会告诉你他还在维护着关键主机列表的host文件，并且通过nsswitch.conf来把它当作备份，在DNS出问题的时候作为替代服务。

复的主机名和IP地址（这往往会导致TCP/IP网络的问题，除非你是在用它来实施虚拟主机、多宿主主机或者高可用性等）。在处理与网络相关的数据的时候，一个好习惯是利用每个机会发现其中的问题。这样可以更早捕获潜在问题，而不至于等到错误蔓延开来的时候才学习教训。因为这个主题很重要，所以本章稍后还会再次讨论。

生成host文件

现在让我们转入更加有趣的主题，那就是生成host文件。假定现在有如下的网络主机数据库文件：

```
name: shimmer
address: 192.168.1.11
aliases: shim shimmy shimmydoodles
owner: David Davis
department: software
building: main
room: 909
manufacturer: Sun
model: M4000
---
name: bendir
address: 192.168.1.3
aliases: ben bendoodles
owner: Cindy Coltrane
department: IT
building: west
room: 143
manufacturer: Apple
model: Mac Pro
---
name: sulawesi
address: 192.168.1.12
aliases: sula su-lee
owner: Ellen Monk
department: design
building: main
room: 1116
manufacturer: Apple
model: Mac Pro
---
name: sander
address: 192.168.1.55
aliases: sandy micky mickydoo
owner: Alex Rollins
department: IT
building: main
room: 1101
manufacturer: Dell
model: Optiplex 740
---
```

这个文件的格式很简单：fieldname:value，并且使用--作为记录分隔符。你可能会发现自己需要更多的字段用来记录其他信息，甚至有可能多到无法用一个平面文件来维护。但是别担心，我们这里介绍的概念是不受后端系统限制的，比如可以从LDAP目录系统（请参考第9章）产生host文件。

下面的代码能分析以上配置文件并生成host文件：

```
my $datafile = 'database';
my $recordsep = "---\n";

open my $DATAFILE, '<', "$datafile" or die "Unable to open datafile:$!\n";

{
    local $/ =
        $recordsep;    # 准备每次从数据库文件读入一条记录

    print "\n# host file - GENERATED BY $0\n# DO NOT EDIT BY HAND!\n#\n";

    my %record;
    while (<$DATAFILE>) {
        chomp;          # 去除每条记录间的分隔符

        # 分割为 key1,value1…… 没错，然后以此构造哈希
        %record = split /\s*/\n/;
        print "$record{address}\t$t$record{name} $record{aliases}\n";
    }
    close $DATAFILE;
}
```

这是输出：

```
#
# host file - GENERATED BY createhosts
# DO NOT EDIT BY HAND!
#
192.168.1.11    shimmer shim shimmy shimmydoodles
192.168.1.3    bendir ben bendoodles
192.168.1.12   sulawesi sula su-lee
192.168.1.55   sander sandy micky mickydoo.
```

现在你该对“系统管理数据库”有些兴趣了吧？

在第3章里，我大力提倡使用一个专门的管理数据库来跟踪账户信息。这个主意也可以用于网络主机数据。这一章我们会展示这个看似简单的平面文件主机数据库可以用来实现令人惊讶的功能。对一个大型网络来说，可能需要使用一个真正的数据库。不过如果想知道这个程序如何运行，请先看看“改善host文件输出”一节末尾的输出。

主机数据库这个想法的提出有几个理由。首先，只要改变一个中心文件并且运行某些脚本就可以完成所有网络服务配置信息的同步修改。毕竟这些生成的配置文件比起手动修改的文件更不容易遗留细微的错误（如缺少分号或者忘记注释符号）。只要我们在编程时注意数据质量，就可以在分析阶段捕获大多数常见问题。

如果你目前仍然看不出这个想法的好处，你会在本章末尾有所改观。

让我们看看这段代码中展示的那些有趣的Perl技巧。首先我们做了一个特殊的设置，在这一小段代码中把\$/设置为一个特殊的值，这样Perl在读取文本的时候会自动把--\n作为记录的分隔标记。这意味着while语句一次循环就能读取整条记录到\$_变量中。我们把local语句放在代码块中，这样变量的修改不会影响块以外的其他代码，将来维护起来也轻松很多。

另外一个有趣的技巧是split()赋值的方式。我们的目标是要把每条记录的内容都按照字段名和值的方式存入哈希。稍后你会看到这其实并非这么简单。不过在这里我们只是用split()来把\$_的内容切割成小的组件。split()返回的数组如表5-1所示。

表5-1：split()返回的数组

元素	值
0	name
1	shimmer
2	address
3	192.168.1.11
4	Aliases
5	shim shimmy shimmydoodles
6	Owner
7	David Davis
8	Department
9	Software
10	Building
11	Main
12	Room
13	909
14	Manufacturer
15	Sun
16	Model
17	M4000

仔细看看这个列表的内容，你会发现我们的数组是按照键-值对的方式排序的（键=Name，值=shimmer，键=Address，值=192.168.1.11等）。这样我们就可以直接把数组赋值成哈希，然后就可以打印需要的内容。

在host文件生成过程中的错误检查

生成host文件是我们能做的第一件事。在使用专用数据库的时候，一个很大的好处在于可以在转换过程中加入错误检查。如同前面介绍的，这样可以避免错误的输入蔓延到生产系统。下面就是在程序中追加了错误检查后的代码：

```
my $datafile = 'database';
my $recordsep = "--\n";

open my $DATAFILE, '<', "$datafile" or die "Unable to open datafile:$!\n";

{
    local $/ =
        $recordsep;    # 准备每次从数据库文件读入一条记录

    print "#\n# host file - GENERATED BY $0\n# DO NOT EDIT BY HAND!\n#\n";

    my %record;
    my %addr;
    while (<$DATAFILE>) {
        chomp;          # 去除每条记录间的分隔符

        # 分割为 key1,value1,... 没错，然后以此构造哈希
        %record = split /\s*\n/;

        # 检查错误的主机名
        if ( $record{name} =~ /[^-.a-zA-Z0-9]/ ) {
            warn "!!!! $record{name} has illegal host name characters, "
                . "skipping...\n";
            next;
        }

        # 检查错误的别名
        if ( $record{aliases} =~ /[^-.a-zA-Z0-9\s]/ ) {
            warn "!!!! $record{name} has illegal alias name characters, "
                . "skipping...\n";
            next;
        }

        # 检查是否缺少地址
        if ( !$record{address} ) {
            warn "!!!! $record{name} does not have an IP address, "
                . "skipping...\n";
            next;
        }

        # 检查是否存在重复的地址
```



```

        if ( defined $addrs{ $record{address} } ) {
            warn "!!!! Duplicate IP addr: $record{name} &
                $addrs{$record{address}}, skipping...\n";
            next;
        }
        else {
            $addrs{ $record{address} } = $record{name};
        }

        print "$record{address}\t$record{name} $record{aliases}\n";
    }
    close $DATAFILE;
}

```

改善host文件输出

我们先从第10章引入日志处理，并在转换过程中加入一些分析。我们可以给文件自动产生有用的文件头、注释和记录分隔符。以下的范例输出就是改进后的效果：

```

#
# host file - GENERATED BY createhosts3
# DO NOT EDIT BY HAND!
#
# Converted by David N. Blank-Edelman (dnb) on Sun Jun  8 00:43:24 2008
#
# number of hosts in the design department: 1.
# number of hosts in the software department: 1.
# number of hosts in the IT department: 2.
# total number of hosts: 4
#

# Owned by Cindy Coltrane (IT): west/143
192.168.1.3    bendir ben bendoodles

# Owned by Alex Rollins (IT): main/1101
192.168.1.55   sander sandy micky mickydoo

# Owned by Ellen Monk (design): main/1116
192.168.1.12   sulawesi sula su-lee

# Owned by David Davis (software: main/909
192.168.1.11   shimmer shim shimmy shimmydoodles

```

这是改进之后的代码，并且加上了一些注释：

```

my $datafile = 'database';
my $recordsep = "--\n";

# 获得 Windows 或 Unix上的用户名
my $user =
    ( $^O eq 'MSWin32' ) ? $ENV{USERNAME} :
    (getpwuid($<))[6] . ' (' . (getpwuid($<))[0] . ')';

```

```

open my $DATAFILE, '<', "$datafile" or die "Unable to open datafile:$!\n";

my %addrs;
my %entries;
{
    local $/ = $recordsep;    # 准备每次从数据库文件读入一条记录

    while (<$DATAFILE>) {
        chomp;                # 去除每条记录间的分隔符
                                # 分割为 key1,value1
        my @record = split /\s*\n/;

        my $record = {};      # 创建空哈希的引用
        %{$record} = @record; # 用 @record 数据填充该哈希

        # 检查错误的主机名
        if ( $record->{name} =~ /^[^-.a-zA-Z0-9]/ ) {
            warn '!!!! '
                . $record->{name}
                . " has illegal host name characters, skipping...\n";
            next;
        }

        # 检查错误的别名
        if ( $record->{aliases} =~ /^[^-.a-zA-Z0-9\s]/ ) {
            warn '!!!! '
                . $record->{name}
                . " has illegal alias name characters, skipping...\n";
            next;
        }

        # 检查是否缺少地址
        if ( !$record->{address} ) {
            warn '!!!! '
                . $record->{name}
                . " does not have an IP address, skipping...\n";
            next;
        }

        # 检查是否存在重复的地址
        if ( defined $addrs{ $record->{address} } ) {
            warn '!!!! Duplicate IP addr:'
                . $record->{name} . ' & '
                . $addrs{ $record->{address} }
                . ", skipping...\n";
            next;
        }
        else {
            $addrs{ $record->{address} } = $record->{name};
        }

        $entries{ $record->{name} } = $record;    # 将此记录保存到一个哈希的哈希中
    }
    close $DATAFILE;
}

```

```

}

# 打印漂亮的头文件
print "#\n# host file - GENERATED BY $0\n# DO NOT EDIT BY HAND!\n#\n";
print "# Converted by $user on " . scalar(localtime) . "\n#\n";
# 按部门统计记录总数并汇报
my %depts;
foreach my $entry ( keys %entries ) {
    $depts{ $entries{$entry}->{department} }++;
}
foreach my $dept ( keys %depts ) {
    print "# number of hosts in the $dept department: $depts{$dept}.\n";
}
print '# total number of hosts: ' . scalar( keys %entries ) . "\n#\n#\n";

# 迭代每条主机记录，打印漂亮的注释和记录条目
foreach my $entry ( keys %entries ) {
    print '# Owned by ', $entries{$entry}->{owner}, ' (',
        $entries{$entry}->{department}, "): ", $entries{$entry}->{building}, '/',
        $entries{$entry}->{room}, "\n";
    print $entries{$entry}->{address}, "\t", $entries{$entry}->{name}, ' ',
        $entries{$entry}->{aliases}, "\n\n";
}

```

这个代码范例与之前的那个最明显的不同在于数据的展现方式。因为之前的版本除了打印主机名和IP地址信息以外别无其他功能，所以我们只需要一个简单的%record哈希就可以了。但在这段代码中，我们要借助一个稍微复杂些的数据结构（哈希的哈希），以便于在打印前进行数据分析和梳理。

其实我们也可以给每个字段生成一个单独的哈希（就像在第2章中展示的*needspace*程序那样），但现在的方式有更好的可维护性。如果我们决定在数据库中加入一个serial_number字段，那么解析代码不必有任何改动，只要直接访问新产生的\$record->{serial_number}就可以了。

唯一的问题是引入的Perl语法可能会让你觉得代码变得更加复杂了。

这是程序做的主要事情：仍然像之前的例子那样分析文件。主要区别是我们把每条记录都存入一个新的匿名哈希。匿名哈希不同于普通哈希之处在于，存取它时使用的是引用而非名字。

我们把这个新创建的匿名哈希赋值给主哈希表的某个键，从而构造出哈希的哈希这样的数据结构。在循环结束的时候，%entries对于每个主机名都有一个键，而对应的值则是另一个包含与那些机器关联的所有字段（如IP adress、room等）的单独的新哈希表。

也许你希望输出的文件是按照IP地址排序的？没问题，只要修改下面这一行来引入新的排序例程就好了：

```
foreach my $entry (keys %entries) {
```

改成：

```
foreach my $entry (sort byaddress keys %entries) {
```

并添加：

```
sub byaddress {  
    my @a = split(/\./,$entries{$a}->{address});  
    my @b = split(/\./,$entries{$b}->{address});  
    ($a[0]<=>$b[0]) ||  
    ($a[1]<=>$b[1]) ||  
    ($a[2]<=>$b[2]) ||  
    ($a[3]<=>$b[3]);  
}
```

注意：这是最容易理解的IP地址排序方式，但是也是性能最差的，因为反复进行`split()`操作的代价太高。更好的方法是比较`pack`过的值，这个想法首先是由Uri Guttman和Larry Rosler提出的（http://www.sysarch.com/Perl/sort_paper.html）。Guttman的`Sort::Maker`模块能帮你完成实现这个方法。而Salvador Fandiño García的`Sort::Key`模块也提供了高性能的排序机制。如果你觉得为了排序而安装一个新模块不太值得，那么在网上搜索“sort ip address perl”，应该也能找到不少更高效的方法。

下面的输出已经做好了排序：

```
PROD: Note that in the Word file, the parenthesis following "software" in the third  
line of the code example below was tagged as Hyperlink. I've stripped this out, as  
I'm assuming this tagging was done by mistake, but may want to confirm with AU that no  
additional tagging was intended there. --Tools  
# Owned by Cindy Coltrane (IT): west/143  
192.168.1.3    bendir ben bendoodles  
  
# Owned by David Davis (software): main/909  
192.168.1.11  shimmer shim shimmy shimmydoodles  
  
# Owned by Ellen Monk (design): main/1116  
192.168.1.12  sulawesi sula su-lee  
  
# Owned by Alex Rollins (IT): main/1101  
192.168.1.55  sander sandy micky mickydoo
```

现在输出变得更加好看了。这证明Perl不但可以带来更多专业上的成功，而且也能提高你的审美能力。

引入源代码控制系统

马上我们就会介绍解决IP地址和主机名映射问题的其他方案。但在此之前我先要为我们

的host文件创建过程引入另一个手法，因为这个host文件在整个网络范围内都要使用，文件中的一个错误会影响整个网络，所以很有必要保护好这个文件。我们需要在文件修改出错时能很快找回之前的配置状态。

建立这个“时光穿梭机”最有效的方法是添加源代码控制系统（source code control system）。一般来说它是给开发人员设计的：

- 用来保存对某个重要文件的所有修改记录。
- 用来避免多人同时修改同一个文件（或者文件的部分内容），从而导致其他人的修改被覆盖。
- 用来使文件回到之前的某个版本，从而解决最近修改引入的问题。

这个功能对系统管理来说非常重要。虽然在第148页“在host文件生成过程中的错误检查”一节加入的错误检查代码能避免某种类型的输入错误和语法错误，但它并不能避免人为错误（比如删除某个重要的主机名、给某个主机设置错误的IP地址或者主机名拼写错误等等）。你确实可以尝试避免某种类型的人为错误，但是相信你无法避免所有类型的错误。还记得那句老话么？“愚蠢是不可避免的，因为愚蠢太富有创造力了。”

你可能会认为只需要对初始的数据库编辑过程进行源代码控制就可以了，但其实即使是生成的结果文件也有必要进行同样的处理，这是因为：

效率

对于大型的输入数据集，处理过程可能要花不少时间。如果你的网络出现问题，而你希望尽快恢复正常，那么等待Perl慢慢处理你的输入文件显得不算太明智。而且很有可能到那时连找到Perl也不是那么容易。

缺乏数据库改变控制

如果你决定使用一个真正的数据库引擎来存放数据，那么想要进行源代码控制就不那么容易。你可能需要自行开发可用于数据库编辑过程的改变控制系统。

我中意的源代码控制系统是修订控制系统（Revision Control System, RCS）^[注3]。它有很多对Perl和系统管理员友善的特性：

- 它是跨平台的。GNU RCS 5.7版能运行在大多数Unix、Windows和Mac OS X系统上。
- 它有一个定义得很好的命令行接口。所有功能都可以从命令行启动，哪怕是在那些非常强调GUI的操作系统上。

注3：如果你想要知道为什么我会推荐RCS，而不是那些更加时髦的源代码控制系统（SVN、Git等等），请参考附录E。

- 它很容易学习。对于基本操作来说只有很少的指令需要学习（参考附录E）。
- 它内置了一些关键字。文件中的某些字符串在RCS之下会被自动扩展。比如文件里所有的\$ Date:\$会被替代成文件最后一次载入RCS系统的时间。
- 它是免费的。

GNU版本的RCS的源代码是免费发行的，而且大多数系统上的二进制程序也都可用。源代码可以从<ftp://ftp.gnu.org/gnu/rcs>下载。如果你还没有用过RCS，那么请先花点时间读一读附录E。接下来的介绍假设你已经熟悉了基础的RCS指令。

Craig Freter写了一个面向对象的名叫Rcs的模块来帮助Perl程序使用RCS功能。使用步骤是：

1. 载入此模块。
2. 告诉模块你的RCS命令程序的位置。
3. 创建一个新的Rcs对象，并且告诉它需要操作的文件名。
4. 调用那些必要的对象方法（通常对应于RCS命令的名字一样）。

我们现在可以把这个功能加入host文件生成的代码中。除了与Rcs模块相关的代码以外，我们还做了些其他改动，这样输出不再是之前版本中的指向标准输出（STDOUT），而是进入某个特定文件。这里只展示那些修改过的代码，原始的代码则被省略了：

```
my $outputfile = "hosts.$$"; # 临时输出文件
my $target      = 'hosts';    # 最终要保存转换后的数据的地方
...
open my $OUTPUT, '>', "$outputfile" or
    die "Unable to write to $outputfile: $!\n";

print $OUTPUT "#\n# host file - GENERATED BY $0\n# DO NOT EDIT BY HAND!\n#\n";
print $OUTPUT "# Converted by $user on " . scalar(localtime) . "\n#\n";

...
foreach my $dept ( keys %depts ) {
    print $OUTPUT "# number of hosts in the $dept department: $depts{$dept}.\n";
}
print $OUTPUT '# total number of hosts: ' . scalar( keys %entries ) . "\n#\n";

# 迭代每条主机记录，打印漂亮的注释和记录条目
foreach my $entry ( keys %entries ) {
    print $OUTPUT '# Owned by ', $entries{$entry}->{owner}, ' (',
        $entries{$entry}->{department}, '): ', $entries{$entry}->{building}, '/',
        $entries{$entry}->{room}, "\n";
    print $OUTPUT $entries{$entry}->{address}, "\t", $entries{$entry}->{name},
        ' ', $entries{$entry}->{aliases}, "\n\n";
}
```

```

close $OUTPUT;

use Rcs;
Rcs->bindir('/arch/gnu/bin');

my $rcsobj = Rcs->new;
$rcsobj->file($target);
$rcsobj->co('-l');
rename( $outputfile, $target )
    or die "Unable to rename $outputfile to $target:$!\n";
$rcsobj->ci( '-u',
    '-m'
    . 'Converted by '
    . ( getpwuid($<) )[6] . ' ('
    . ( getpwuid($<) )[0] . ') on '
    . scalar localtime );

```

这段程序假定目标文件起码已经被提交过一次。

要检查这段代码的执行效果，我们可以看看`rlog hosts`命令的输出：

```

revision 1.5
date: 2007/05/19 23:34:16; author: dnb; state: Exp; lines: +1 ?1
Converted by David N. Blank-Edelman (dnb) on Tue May 19 19:34:16 2007
-----
revision 1.4
date: 2007/05/19 23:34:05; author: eviltwin; state: Exp; lines: +1 ?1
Converted by Divad Knaib-Namlede (eviltwin) on Tue May 19 19:34:05 2007
-----
revision 1.3
date: 2007/05/19 23:33:35; author: dnb; state: Exp; lines: +20 ?0
Converted by David N. Blank-Edelman (dnb) on Tue May 19 19:33:16 2007

```

这里没有显示文件的各个版本之间到底有怎样的改变（只是显示修改的行数），但你已经可以看到文件每次的改变都被记录下来。如果需要，我们可以用`rcsdiff`命令来仔细检查变动的细节。在极端情况下，我们的改动可能破坏了网络的稳定性，那时候我们就需要恢复到之前某个版本的能力。

在进一步讨论其他名称服务之前，让我们快速回顾一下目前已经学到的三个技术：

- 从某种外部数据库来生成配置文件是一个重大改进。
- 在数据转换过程中进行错误检查，这样能避免错误输入对整个网络产生影响，这是一个很好的做法。
- 引入源代码控制系统来帮助从复杂错误中快速恢复并跟踪文件修改，这也非常有用。

NIS、NIS+和WINS

Sun Microsystems的开发人员意识到“为每台机器编辑host文件”的做法行不通，所以他们就发明了*Yellow Pages*，简称YP。这个服务用来发布所有网络级别的配置文件，比如*/etc/hosts*、*/etc/passwd*、*/etc/services*等等。在这一章我们会专注于主机名和IP地址映射信息的发布。

YP在1990年被改名为网络信息服务（Network Information Service，NIS）。改名之前，英国电话公司曾经通过律师提出申诉，说他们在英国持有“Yellow Pages”的商标。可是在NIS里面至今仍然可以看到许多YP的痕迹，比如命令名和库程序名，如*ypcat*、*ypmatch*、*yppush*等等。

所有主流的Unix变种都支持NIS。Mac OS X（至少在Tiger和之后的版本中）则力图让客户选择使用Directory Access工具程序，从而逐渐减少对NIS的依赖。这个配置的位置是*/Applications/Utilities*（选中“BSD Flat File and NIS”下面的那个方框，并且点击“Apply”。OS X也带有支持NIS服务的程序（如*/usr/libexec/ypserv*、*/var/yp/**等等），不过我从来没有看到有人使用过。

NIS和Windows之间的关系就比较复杂了。在写本书第一版时，确实是可以把Windows的身份认证库替换成访问NIS服务器的自定义代码，这样就可以替代域验证了。当时的解决方案名字叫做NISGINA。

如果还需要让Windows系统使用NIS系统的数据，最好的解决方案应该是使用Samba（<http://www.samba.org>）作为中介。而在NIS服务器方面，微软也使得他们的Windows 2003 R2产品可以用NIS的方式发布Active Directory的信息给NIS客户机。这个解决方案对那些决定采用Active Directory作为核心的认证系统，但又必须同时用NIS来支持那些非Windows的客户机的人来说是件好事。微软如今把这个组件命名为“Identity Management for Unix”（IdMU）。你可以通过如下方式手动安装这个产品：选择“添加/删除程序”→“添加/删除Windows组件”→“Active Directory服务”，单击“详细信息”。

在NIS中，管理员设置一台或多台机器成为服务器，其他机器可以从它们那里获得服务。只有一台服务器可以成为主服务器，而其他服务器只能成为从服务器。只有主服务器持有实际文本文件（比如*/etc/hosts*或者*/etc/passwd*）的“原件”，所有的修改都是通过主服务器发布到从服务器。

网络中所有的主机在需要主机名到IP地址的映射信息时只需要查询服务器即可，不必再从本地文本文件中搜索。客户机可以从主服务器或者从服务器请求此信息。收到客户机

查询请求之后，服务器会查询NIS映射，这是通过在主服务器上把映射信息转化成DBM格式并发布到从服务器从而产生的文件。转换过程一般需要使用`makedbm`和其他一些工具，细节可以参考`/var/yp`目录下的`Makefile`文件。共享这些NIS映射文件的所有服务器和客户机组成了NIS域。

在NIS中，系统管理变得比较简单。比如现在`oog.org`买了更多的机器，然后希望把它们集成到网络中。这对网络管理员来说，只需要在NIS主服务器上编辑`host`文件，并且把新版本发送到从服务器。这样，NIS域里所有的客户机都会立刻“认识”这台新机器。NIS就是这样提供了简易的管理功能，并同时提供了冗余（一台服务器离线的情况下，其他服务器自动响应客户机请求）和负载均衡（避免了全网络的客户机依赖一台服务器）。

有了以上的基本知识，我们就可以开始介绍Perl是如何帮助我们完成与NIS相关的管理任务了。让我们从把数据放入NIS开始。你可能会很惊讶地发现我们已经完成了。因为可以直接把之前生成的`host`文件放在NIS主服务器的源文件目录下，然后激活常规的发布机制即可，也就是在`/var/yp`目录下调用`make`。在默认情况下，`/var/yp`目录下的`Makefile`会使用主服务器的配置文件来更新NIS映射。

注意：实际应用中往往需要把NIS映射源文件存放在单独的目录下，当然这也需要同步修改`Makefile`。因为这样可以把NIS主服务器的数据和其他NIS域成员的数据分开存放，比如NIS主服务器的`/etc/passwd`文件就可以避免与整个域的机器共享。

更加有趣的任务是通过查询NIS服务器来获取NIS信息。做这件事情最简单的方法是通过使用Rik Harris的（目前维护者是Ed Santiago）`Net::NIS`模块来获取这个信息。

下面的例子展示的是如何通过`Net::NIS`模块来获取并打印整个域的主机映射信息，效果类似于NIS命令`ypcat`：

```
use Net::NIS;

# 取得默认的 NIS 域名
my $domain = Net::NIS::yp_get_default_domain();

# 获取映射
my ( $status, $info ) = Net::NIS::yp_all( $domain, 'hosts.byname' );
foreach my $name ( sort keys %{$info} ) {
    print "$name => $info->{$name}\n";
}
```

我们先查询本地主机的默认域名。有了这个信息，我们就可以调用`Net::NIS::yp_all()`来获取整个域的主机映射。这个函数调用会返回一个状态变量和一个包含映射信息的哈希表引用。我们用Perl常见的反引用语法来打印信息。

如果我们只查询某台主机的IP地址，就能更有效地查询服务器：

```
use Net::NIS;

my $hostname = 'olaf.oog.org';

my $domain = Net::NIS::yp_get_default_domain();
my ( $status, $info ) =
    Net::NIS::yp_match( $domain, 'hosts.byname', $hostname );

print "$info\n";
```

`Net::NIS::yp_match()`返回一个状态变量和相应的标量值（包含查询的结果信息）。

如果`Net::NIS`模块无法在你的机器上编译或运行，还可以考虑使用“调用外部程序”的方法来解决：

```
@hosts=`<path to>/ypcat hosts`
```

或者这样做：

```
open my $YPCAT, '-|', '<path to>/ypcat hosts';
while (<$YPCAT>){...}
```

在本节末尾，让我们结合两种方法（使用`Net::NIS`和外部程序调用）来完成一个任务。这段简单的代码能够列出NIS域的所有服务器，并且使用`ypoll`程序逐个进行查询。如果某个服务器在响应查询时出错，我们就能立刻发现问题：

```
use Net::NIS;

my $ypolllex = '/usr/sbin/ypoll';    # 可执行程序 ypoll 的路径

my $domain = Net::NIS::yp_get_default_domain();    # 我们的 NIS 域

my ( $status, $info ) = Net::NIS::yp_all( $domain, 'ypservers' );

foreach my $server ( sort keys %{ $info } ) {
    my $answer = `ypolllex -h $server hosts.byname`;
    if ( $answer !~ /has order number/ ) {
        print STDERR "$server is not responding properly!\n";
    }
}
```

我们可以进一步改进这段代码（比如检查返回顺序是否与服务器的一致），但这个挑战是留给那些有兴趣的读者的。

NIS+

Sun还开发了NIS的下一代产品，名字叫做NIS+，并且在Solaris操作系统中集成了这个

产品。NIS+主要是为了解决NIS里面较为严重的问题，尤其是安全问题。但很不幸，NIS+没有被Unix世界接受，不像它的前任NIS那么普及。不过从某种角度来说这也是幸运的，因为NIS+的管理实在有些麻烦。到目前为止，只有Sun公司生产的机器才能得到NIS+的支持。Thorsten Kukuk尝试把NIS+移植到Linux的努力 (<http://www.linux-nis.org/nisplus/>) 已经终止了。Sun公司也不建议使用它，而是推荐使用LDAP。考虑到这样的趋势，我们不会在本书中继续讨论NIS+，不过如果你有兴趣的话，可以看看NIS+的Perl模块Net::NISPlus，作者是Harris。

Windows Internet 名称服务 (WINS)

我们再来看一个逐渐被淘汰的老协议。当时微软独有的网络协议NetBIOS over TCP/IP (NetBT) 也需要处理机器名到IP地址的映射问题。微软开始的解决方案是用`lmhosts`文件，可以说它是对标准`host`文件的一种包装。紧接着这个解决方案被NIS类型的机制代替了，在NT 3.5版本以后，微软开始推广一个名为Windows Internet名称服务 (Windows Internet Name Server, WINS) 的方案 (scheme)。WINS和NIS主要的差别如下所示：

- WINS专注于发布机器名到IP地址的映射信息，不像NIS还同时发布其他信息（比如密码文件、网络信息、端口映射和用户组信息等）。
- WINS服务器主要是通过接收客户端的注册信息来获取信息的（当然也支持服务器端预先配置）。在它们获取了手动配置的IP地址或者DHCP分配的地址以后，WINS客户端需要向服务器注册（或者重新注册）这个信息。这与NIS的模式不同，WINS中的客户机只是单向查询服务器中预先配置好的信息，不会对服务器端信息进行修改，只有密码是需要向服务器注册的。

WINS和NIS类似，通过主从服务器的配置方式提供了多服务器的备份和负载均衡机制。在Windows 2000之后，WINS的使用不再常见，取而代之的是动态域名服务 (Dynamic Domain Name Service, DDNS)，这是一个对基础DNS系统的种扩展，我们很快会介绍它。

考虑到WINS和NIS+一样快要成为历史名词，我们不会介绍如何用Perl来操纵它。目前并没有很好的办法通过Perl与WINS直接交互（至少我不知道哪个模块是专门为WINS设计的）。如果你需要支持这个协议，最好的方法应该是从命令行调用Windows资源工具包里面的工具程序（如WINSCHK和WINSCL）。

域名服务 (DNS)

尽管NIS和WINS一度非常有用，但内在的设计缺陷直接导致它们无法应用到整个因特网。主要原因有两个：

扩展性

尽管这两个方案都支持多服务器运行，但每个服务器都必须存储整个网络的映射信息^[注4]。这些信息必须完整地复制到所有服务器，而这个过程必定非常耗时。另外，WINS还饱受动态注册机制的困扰，足够数量的WINS客户机注册信息就能把一台WINS服务器拖垮。

管理控制

之前我们虽然讨论了一些技术层面的问题，但却没有谈到资源管理问题。NIS在管理上要求单点控制，域的主服务器必须完全控制整个域的映射信息，任何改动都必须在主服务器上完成（并发布）。这就导致它不适合在因特网级别使用。

域名服务（Domain Name Service, DNS）的发明正是为了解决host文件、NIS、NIS+和WINS在管理方面的设计缺陷。在DNS中，网络的名称空间被切分成一组“顶级域名”，而每个顶级域名还可以被切分成子域名，子域名又可以进一步切分，以此类推，针对每一级子域名，都可以单独设定不同配置。正是凭借着这一点，我们才能对域名的控制和管理进一步细化。

网络客户机会根据自己所在的位置查询最接近的域名服务器。如果在服务器本地就能找到所查询的主机名，那么结果就直接返回给客户机。而在大多数网络上，收到的域名查询请求都是关于本域的主机的，所以不必请求查询远程服务器就能找到结果。这样的设计使得可扩展性得到了增强。另外还可以为了冗余和负载均衡的需要，设立多台DNS服务器，称作辅助服务器（secondary server）或从服务器（slave server）。

如果DNS服务器收到了关于未知域名的查询，那么它既可以让客户机向其他（往往是更高级别）的服务器发起查询，也可以代替客户机向其他服务器发起查询。

在这个方案中，没有任何一台服务器需要知道整个网络的拓扑结构，而且大多数查询都是本地处理的，本地域名信息也是本地管理的。这就是为什么DNS能够最终胜出，让系统管理员和用户都满意，这也是为什么NIS和WINS都力图集成DNS。比如，Solaris的NIS服务器就能配置成自动用DNS查询未知主机名。查询结果是以标准NIS查询回复返回的，所以客户机并不需要知道幕后DNS查询的存在。同样地，微软的DNS服务器也有类似的配置功能，在收到的查询请求包含未知主机名的时候，它可以自动查询WINS服务器并返回结果。

生成DNS（BIND）配置文件

我们可以采用类似于生成host文件和NIS源文件那样的机制来生成DNS配置文件。这里主要是针对比较常见的BIND DNS服务器（<http://www.isc.org/software/bind>）：

注4： NIS+提供了一种机制来允许域外的客户机查询地址映射信息，但不如DNS的可扩展性强。

- 我们使用专门的数据库来存放原始数据（可以把本书所有的数据库合并为一个）。
- 用合适的输出格式转换数据，并且在转换过程中进行错误检查。
- 使用RCS或其他源码控制系统来跟踪文件的历史版本。

对于DNS来说，我们必须扩展第二步，因为转换过程更加复杂。在进一步深入介绍的时候，你可能会发现Paul Albitz和Cricket Liu合著的《DNS and BIND》(O'Reilly)是本非常好的参考书籍。

创建管理性文件头

DNS配置文件都有一个管理性文件头，用来声明它所代表的服务器和相关信息。这个文件头最重要的部分就是起始授权机构（Start of Authority，SOA）记录了。它一般包含：

- 这个DNS服务器负责的管理域的名称
- 管辖域内主DNS服务器的名称
- DNS管理员的联系方式
- 配置文件的序列号（稍后介绍）
- 从服务器的自动刷新和重试间隔（也就是说何时与主服务器同步）
- 数据的生存时间（Time to Live，TTL）设定（也就是信息可以安全缓冲的时间长度）

以下就是样本文件头：

```
@ IN SOA  dns.oog.org. hostmaster.oog.org. (
                                2007052900 ; serial
                                10800      ; refresh
                                3600       ; retry
                                604800    ; expire
                                43200)    ; TTL

@                IN  NS  dns.oog.org.
```

这里的大多数信息在每次生成文件时都是固定不变的。唯一需要注意的信息是序列号。每隔X秒钟（这个长度取决于自动刷新闻隔）从服务器会尝试连接主服务器，并同步更新它们的DNS数据。现在较新的从服务器（如BIND v8+或者微软DNS）可以智能判断何时需要更新数据。两种服务器都会检查主服务器的SOA记录。如果SOA中的序列号比当前的序列号更高，那么就会启动区域传送（也就是从服务器下载新数据集）。因为上述的同步机制依赖于序列号的增长，所以每次创建DNS配置文件之后都应该递增这个数字。要知道，有很多DNS问题就是由于序列号的更新错误导致的。

至少有两种方法能保证这个序列号总是增长的：

- 读取以前的配置文件并增加其中的值。
- 从某个能确保增长的外部数据源（比如系统时钟或者文件的RCS版本号）计算出新的值。

下面的范例代码就同时使用了以上两种方法来产生一个DNS区域文件（zone file）。它采用了Albitz和Liu的书中推荐的格式来创建序列号（YYYYMMDDXX，其中YYYY是年份，MM是月份，DD是日期，而XX是一个两位的计数器，用来标识一天内的多次变化）：

```
# 取得今天的日期并表示为YYYYMMDD的形式
my @localtime = localtime;
my $today      = sprintf( "%04d%02d%02d",
    $localtime[5] + 1900,
    $localtime[4] + 1,
    $localtime[3] );

# 取得 Windows 或 Unix 上的用户名
my $user =
( $^O eq 'MSWin32' )
? $ENV{USERNAME}
: ( getpwuid($<) )[6] . ' (' . ( getpwuid($<) )[0] . ' )';

sub GenerateHeader {
    my ($olddate,$count);

    # 尝试打开旧文件并读取原来的序列号
    # （假设旧文件采用此种格式）
    if ( open( my $OLDZONE, '<', $target ) ) {
        while (<$OLDZONE>) {
            last if ( $olddate, $count ) = /(\d{8})(\d{2})*.serial/;
        }
        close $OLDZONE;
    }

    # 如果已经定义过变量 $count，就说明找到了原来用过的序列号。
    # 如果原来的序列号属于今天，则递增最后两位数字；
    # 否则生成今天的第一个序列号。
    my $count = ( defined $count and $olddate eq $today ) ? $count + 1 : 0;
    my $serial = sprintf( "%8d%02d", $today, $count );

    # 准备头部信息
    my $header = "; dns zone file - GENERATED BY $0\n";
    $header .= "; DO NOT EDIT BY HAND!\n;\n";
    $header .= "; Converted by $user on " . scalar( localtime ) . "\n;\n";

    # 统计各部门记录数量并汇报
    foreach my $entry ( keys %entries ) {
        $depts{ $entries{$entry}->{department} }++;
    }
    foreach my $dept ( keys %depts ) {
        $header .=
            "; number of hosts in the $dept department: " . "$depts{$dept}.\n";
    }
}
```

```

$header .=
'; total number of hosts: ' . scalar( keys %entries ) . "\n;\n\n";

$header .= <<"EOH";

@ IN SOA  dns.oog.org. hostmaster.oog.org. (
    $serial ; serial
    10800   ; refresh
    3600    ; retry
    604800  ; expire
    43200)  ; TTL

@
    IN NS  dns.oog.org.

EOH

return $header;
}

```

我们的代码尝试读取以前的DNS配置文件以获取上一次的序列号，然后把它切分成为日期和计数器字段。如果读取的日期和当前日期一致，则递增计数器值；否则使用当前日期加上初始计数器值00作为序列号。确定序列号之后，剩下的代码就很简单了，只是一个格式化的文件头打印程序而已。

生成多个配置文件

刚才我们介绍了生成DNS配置文件的文件头的方法，是时候介绍更重要的部分了。一个配置良好的DNS服务器都有域内（或者区域内）的正向和逆向地址映射信息。所以每个区域至少需要两个配置文件。保证这两个文件的数据同步更新的最好方法就是同时创建它们。

这是我们本章的最后一个文件生成脚本，所以让我们把之前介绍的技巧汇总起来。我们的脚本会从一个数据库文件获取信息，生成必要的DNS区域配置文件。

要让这个脚本保持简单易懂，我们只能对数据作出一些限制，把注意力集中在网络的拓扑结构和名称空间上。这个脚本假定网络中的子网是C类的并且只有一个DNS区域。这样我们只需要创建一个正向映射文件和对应的逆向映射文件。可以在需要的时候扩展这个例子来支持更多的子网和区域（通过多个文件来支持）。

下面就是我们的代码要完成的任务概要：

1. 读取数据库文件，生成哈希的哈希，检查数据的合法性。
2. 生成文件头。
3. 生成正向映射文件（主机名到IP地址的映射）并且提交到RCS。
4. 生成逆向映射文件（IP地址到主机名的映射）并且提交到RCS。

下面是代码和相应的输出：

```
use Rcs;

my $datafile = 'database';      # our host database
my $outputfile = "zone.$$";     # our temporary output file
my $target = 'zone.db';        # our target output
my $revtarget = 'rev.db';       # our target output for the reverse mapping
my $defzone = '.oog.org';       # the default zone being created
my $rcsbin = '/usr/local/bin';  # location of our RCS binaries
my $recordsep = "--\n";

# 取得今天的日期并表示为 YYYYMMDD 的形式
my @localtime = localtime;
my $today = sprintf( "%04d%02d%02d",
    $localtime[5] + 1900,
    $localtime[4] + 1,
    $localtime[3] );

# 取得 Windows 或 Unix 上的用户名
my $user =
    ( $^O eq 'MSWin32' )
    ? $ENV{USERNAME}
    : ( getpwuid($<) )[6] . ' ( ' . ( getpwuid($<) )[0] . ' )';

# 读取数据库文件
open my $DATAFILE, '<', "$datafile" or die "Unable to open datafile:$!\n";

my %addrs;
my %entries;
{
    local $/ = $recordsep;      # 准备每次从数据库文件读入一条记录

    while (<$DATAFILE>) {
        chomp;                  # 去除每条记录间的分隔符
                                # 分割为 key1,value1
        my @record = split /\s*\|/;

        my $record = {};        # 创建空哈希的引用
        %{$record} = @record;    # 用 @record 数据填充该哈希

        # 检查错误的主机名
        if ( $record->{name} =~ /^[^-.a-zA-Z0-9]/ ) {
            warn '!!!! '
                . $record->{name}
                . " has illegal host name characters, skipping...\n";
            next;
        }

        # 检查错误的别名
        if ( $record->{aliases} =~ /^[^-.a-zA-Z0-9\s]/ ) {
            warn '!!!! '
                . $record->{name}
                . " has illegal alias name characters, skipping...\n";
            next;
        }
    }
}
```



```

    }

    # 检查是否缺少地址
    if ( !$record->{address} ) {
        warn '!!!! '
        . $record->{name}
        . " does not have an IP address, skipping...\n";
        next;
    }

    # 检查是否存在重复的地址
    if ( defined $addrs{ $record->{address} } ) {
        warn '!!!! Duplicate IP addr:'
        . $record->{name} . ' & '
        . $addrs{ $record->{address} }
        . ", skipping...\n";
        next;
    }
    else {
        $addrs{ $record->{address} } = $record->{name};
    }

    $entries{ $record->{name} } = $record;    # 添加到哈希的哈希
}
close $DATAFILE;
}

my $header = GenerateHeader();

# 创建正向映射文件
open my $OUTPUT, '>', "$outputfile"
or die "Unable to write to $outputfile:$!\n";
print $OUTPUT $header;

foreach my $entry ( sort byaddress keys %entries ) {
    print $OUTPUT "; Owned by ", $entries{$entry}->{owner}, ' (',
        $entries{$entry}->{department}, "): ", $entries{$entry}->{building}, '/',
        $entries{$entry}->{room}, "\n";

    # 打印 A 记录
    printf $OUTPUT "%-20s\tIN A      %s\n", $entries{$entry}->{name},
        $entries{$entry}->{address};

    # 打印所有的 CNAMEs (别名)
    if ( defined $entries{$entry}->{aliases} ) {
        foreach my $alias ( split( ' ', $entries{$entry}->{aliases} ) ) {
            printf $OUTPUT "%-20s\tIN CNAME %s\n", $alias,
                $entries{$entry}->{name};
        }
    }
    print $OUTPUT "\n";
}

close $OUTPUT;

```

```

Rcs->bindir($rcsbin);
my $rcsobj = Rcs->new;
$rcsobj->file($target);
$rcsobj->co('-l');
rename( $outputfile, $target )
    or die "Unable to rename $outputfile to $target:!\n";
$rcsobj->ci( '-u', '-m' . "Converted by $user on " . scalar(localtime) );

# 现在创建逆向映射文件
open my $OUTPUT, '>', "$outputfile"
    or die "Unable to write to $outputfile:!\n";
print $OUTPUT $header;
foreach my $entry ( sort byaddress keys %entries ) {
    print $OUTPUT ' '; Owned by ', $entries{$entry}->{owner}, ' (',
        $entries{$entry}->{department}, '): ', $entries{$entry}->{building}, '/',
        $entries{$entry}->{room}, "\n";

    # 这里使用脚本开头定义的默认区域
    printf $OUTPUT "%-3d\tIN PTR      %s$defzone.\n\n",
        ( split /\./, $entries{$entry}->{address} )[3], $entries{$entry}->{name};
}

close $OUTPUT;
$rcsobj->file($revtarget);
$rcsobj->co('-l'); # 假设目标至少已经被检查过一次
rename( $outputfile, $revtarget )
    or die "Unable to rename $outputfile to $revtarget:!\n";
$rcsobj->ci( "-u", "-m" . "Converted by $user on " . scalar(localtime) );

sub GenerateHeader {
    my ( $olddate, $count );

    # 尝试打开旧文件并读取原来的序列号
    # (假设旧文件采用此种格式)
    if ( open( my $OLDZONE, '<', $target ) ) {
        while (<$OLDZONE) {
            last if ( $olddate, $count ) = /\d{8}\d{2}.*serial/;
        }
        close $OLDZONE;
    }

    # 如果已经定义过变量 $count, 就说明找到了原来用过的序列号。
    # 如果原来的序列号属于今天, 则递增最后两位数字;
    # 否则生成今天的第一个序列号。
    my $count = ( defined $count and $olddate eq $today ) ? $count + 1 : 0;
    my $serial = sprintf( "%8d%02d", $today, $count );

    # 准备头部信息
    my $header = "; dns zone file - GENERATED BY $0\n";
    $header .= "; DO NOT EDIT BY HAND!\n;\n";
    $header .= "; Converted by $user on " . scalar( (localtime) ) . "\n;\n";

    # 统计各部门记录数量并汇报
    my %depts;

```

```

foreach my $entry ( keys %entries ) {
    $depts{ $entries{$entry}->{department} }++;
}
foreach my $dept ( keys %depts ) {
    $header .=
        "; number of hosts in the $dept department: " . "$depts{$dept}.\n";
}
$header .=
    "; total number of hosts: " . scalar( keys %entries ) . "\n;\n\n";

$header .= <<"EOH";

@ IN SOA    dns.oog.org. hostmaster.oog.org. (
                $serial ; serial
                10800    ; refresh
                3600     ; retry
                604800   ; expire
                43200)   ; TTL

@
                IN NS   dns.oog.org.

EOH

return $header;
}

sub byaddress {
    my @a = split( /\./, $entries{$a}->{address} );
    my @b = split( /\./, $entries{$b}->{address} );
    ( $a[0] <=> $b[0] )
    || ( $a[1] <=> $b[1] )
    || ( $a[2] <=> $b[2] )
    || ( $a[3] <=> $b[3] );
}

```

下面就是生成的正向映射文件 *zone.db* 的内容:

```

; dns zone file - GENERATED BY createdns
; DO NOT EDIT BY HAND!
;
; Converted by David N. Blank-Edelman (dnb); on Fri May 29 15:46:46 2007
;
; number of hosts in the design department: 1.
; number of hosts in the softwaredepartment: 1.
; number of hosts in the IT department: 2.
; total number of hosts: 4
;

@ IN SOA    dns.oog.org. hostmaster.oog.org. (
                2007052900 ; serial
                10800     ; refresh
                3600      ; retry
                604800    ; expire
                43200)    ; TTL

```

```

@                               IN NS  dns.oog.org.

; Owned by Cindy Coltrane (marketing): west/143
bendir           IN A      192.168.1.3
ben              IN CNAME  bendir
bendoodles       IN CNAME  bendir

; Owned by David Davis (software): main/909
shimmer          IN A      192.168.1.11
shim             IN CNAME  shimmer
shimmy           IN CNAME  shimmer
shimmydoodles    IN CNAME  shimmer

; Owned by Ellen Monk (design): main/1116
sulawesi         IN A      192.168.1.12
sula             IN CNAME  sulawesi
su-lee           IN CNAME  sulawesi

; Owned by Alex Rollins (IT): main/1101
sander           IN A      192.168.1.55
sandy            IN CNAME  sander
micky            IN CNAME  sander
mickydoo         IN CNAME  sander

```

下面是生成的逆向映射文件*rev.db*的内容:

```

; dns zone file - GENERATED BY createdns
; DO NOT EDIT BY HAND!
;
; Converted by David N. Blank-Edelman (dnb); on Fri May 29 15:46:46 2007
;
; number of hosts in the design department: 1.
; number of hosts in the softwaredepartment: 1.
; number of hosts in the IT department: 2.
; total number of hosts: 4
;

@ IN SOA  dns.oog.org. hostmaster.oog.org. (
                                2007052900 ; serial
                                10800      ; refresh
                                3600       ; retry
                                604800     ; expire
                                43200      ; TTL
)

@                               IN NS  dns.oog.org.

; Owned by Cindy Coltrane (marketing): west/143
3  IN PTR  bendir.oog.org.

; Owned by David Davis (software): main/909
11 IN PTR  shimmer.oog.org.

; Owned by Ellen Monk (design): main/1116

```

```
12 IN PTR      sulawesi.oog.org.  
; Owned by Alex Rollins (IT): main/1101  
55 IN PTR      sander.oog.org.
```

这种创建文件的方法带来了进一步扩展的机会。目前我们生成的文件都是来自于一个文本文件数据库。我们做的只是读取来自数据库的记录并产生目标文件，也许加上一些格式改进。除此之外生成的文件中并无其他外来信息。

有时候在生成文件的过程中加入一些信息也很有用。比如在DNS配置文件中，如果需要的话，可以让脚本完成额外的MX（Mail eXchange,邮件交换）记录生成。这样可以给所有的主机配置一个中央邮件服务器。需要修改的代码很少：

```
# 打印A记录  
printf $OUTPUT "%-20s\tIN A      %s\n",  
    $entries{$entry}->{name},$entries{$entry}->{address};
```

改成：

```
# 打印A记录  
printf $OUTPUT "%-20s\tIN A      %s\n",  
    $entries{$entry}->{name},$entries{$entry}->{address};  
  
# 打印MX记录  
print $OUTPUT "                IN MX 10 $mailserver\n";
```

这样就能让\$mailserver这台主机负责接收域中所有主机的邮件。如果那台机器真的具备处理邮件的能力，那么这行简单的Perl代码就好比给我们构造了一个有用的基础设施组件（即集中邮件处理）。

DNS检查：迭代方式

我们已经花了不少时间介绍如何创建配置文件来支持各种网络名称服务，但这只是系统管理任务的一方面而已。运营良好的网络必须有一个服务检查机制，这样才能更好地支持整体服务的运行。

对于系统管理员或者网络管理员来说，一个重要的问题就是：“我们所有的DNS服务器都在正常运行么？”而在定位问题时，另一个重要的问题是：“所有的服务器提供的信息都一致么？”或者更加精确地说：“对同样的查询，所有的服务器都能返回同样的结果么？它们之间的信息是不是同步的？”我们在这一节里会尝试解答这些问题。

在第2章中我们实际印证了Perl的座右铭“条条大路通罗马”。Perl本身的TMTOWTDI特色使得它非常善于作为“迭代式开发”的原型开发语言。这种渐进演化的任务型开发模式对于系统管理程序（或其他程序）来说非常合适。对Perl来说，快速开发出简易脚本

来完成任务是很容易的。当然也可以在之后的某个时刻对脚本进行改进，使它更为精致。甚至有可能进行第三次改进，这种改进当然也可以是对问题的另一种解决。

在这一节，我们会分析对DNS进行一致性检查的三种方法。这三种方法是按照解决问题时会实际尝试的次序罗列的。这个次序也展示了Perl是如何帮助我们改进解决方案的，不过每个人的次序会有些差异。第三种解决方案，使用Net::DNS模块可能是最容易也最少出错的，但它并不适用于所有情况，所以我们会从最“稚嫩”的尝试开始分析。请特别注意在每个方法之后展开的优劣分析。

下面是任务的描述：要求Perl脚本查询某个主机名在一组DNS服务器上的解析结果，看看它们返回的值是否一致。为了简化问题，我们假定测试的主机只有一个静态IP地址（也就是说没有多个网卡或者绑定多个地址）。

在我们逐个介绍每种解决方案之前，先展示我们将使用的“驱动”代码：

```
use Data::Dumper;

my $hostname = $ARGV[0];
my @servers = qw(nameserver1 nameserver2 nameserver3); # 域名服务器

my %results;
foreach my $server (@servers) {
    $results{$server}
        = LookupAddress( $hostname, $server );    # 把查询结果存入 %results
}

my %inv = reverse %results;    # 将结果哈希倒序
if (scalar keys %inv > 1) {    # 看看其中有多少元素
    print "There is a discrepancy between DNS servers:\n";
    print Data::Dumper->Dump( [ \%results ], ['results'] ), "\n";
}
```

对于@servers列表中的每个DNS服务器，我们调用LookupAddress()子例程返回查询结果。这个子例程每次查询一台DNS服务器，并且把返回的IP地址和DNS服务器的名字放在一个名叫%results的哈希中。其中，DNS服务器名是键，而返回的IP地址是值。

要判断%results里面的信息是否一致（也就是说DNS服务器的响应一致）有很多办法。这里我们选择了一个“反转”哈希赋值的方法，把%results赋值给另外一个哈希，并且在赋值时对调键和值。对这种方法来说，如果%results里面所有的值是一样的，那么产生的哈希就应该只有一个键。如果键的数量不止一个，那么我们就知道遇到了问题，所以就可以调用Data::Dumper->Dump()来漂亮地显示%results的内容供系统管理员分析。

以下就是样本错误信息：

```

There is a discrepancy between DNS servers:
$results = {
    nameserver1 => '192.168.1.2',
    nameserver2 => '192.168.1.5',
    nameserver3 => '192.168.1.2',
};

```

现在让我们来看看LookupAddress()子例程实现时的备选方案。

使用nslookup

如果你有Unix背景，或者你曾经使用过其他脚本语言，那么你可能会首先尝试类似shell脚本的解决方案。下面的代码展示了如何使用Perl脚本调用外部程序来完成这个任务：

```

use Data::Dumper;

my $hostname = $ARGV[0];

my @servers = qw(nameserver1 nameserver2 nameserver3 nameserver4);

my $nslookup = '/usr/bin/nslookup';

my %results;
foreach my $server (@servers) {
    $results{$server}
        = LookupAddress( $hostname, $server );    # 把查询结果存入 %results
}

my %inv = reverse %results;    # 将结果哈希倒序
if ( scalar keys %inv > 1 ) {    # 看看其中有多少元素
    print "There is a discrepancy between DNS servers:\n";
    print Data::Dumper->Dump( [ \%results ], ['results'] ), "\n";
}

sub LookupAddress {
    my ( $hostname, $server ) = @_;
    my @results;

    open my $NSLOOK, '-|', "$nslookup $hostname $server"
        or die "Unable to start nslookup:!\n";

    while (<$NSLOOK>) {
        next until (/^Name: /);    # 忽略不是 "Name: " 开头的行
        chomp( $result = <$NSLOOK> );    # 之后一行是 Address: 响应
        $result =~ s/Address(es)?:\s+//; # 去掉标签
        push( @results, $result );
    }
    close $NSLOOK;
    return join( ' ', sort @results );
}

```

这个方法的优点：

- 代码写起来简单快速，几乎是从shell脚本逐行翻译过来的。
- 不必进行复杂的网络编程。
- 这个方法是Unix最经典的完成任务的方式：使用通用语言来黏合特殊工具程序，而不是开发一个完整的大型程序。
- 当你无法在Perl中编码客户机/服务器通信时这可能是唯一可行的方法。比如说，如果某个服务器被配置为必须由特殊客户端来访问，这时用Perl来编写网络客户端就不行了。

这个方法的缺陷：

- 它依赖于外部程序。如果那个程序不可用或者突然改变了输出格式怎么办呢？
- 它常常很慢，因为在每次查询之前必须先启动另外一个进程。我们可以使用双向管道来连接一个*nslookup*进程，这样可以让它一直运行，并在我们需要的时候调用。这可能会让代码更加复杂，但如果真的要提升代码性能，这个改进还是值得的。
- 我们缺少对细节的控制能力。比如对于服务器超时、查询重试次数和域搜索列表的控制，这些都依赖于*nslookup*的能力（实际是这个进程底层依赖的库程序的能力）。

直接对网络socket编程

如果你是一个“铁血系统管理员”，那么你可能并不喜欢调用别的程序。可能你会考虑直接用Perl来实现DNS查询，通过手动编程构造网络数据包并发送给服务器，然后等待响应并解析输出。

这一节的代码可能是整本书里面最复杂的，它是参照后面列出的资料编写的，还参考了包括Michael Fuhr和Olaf Kolkman的模块（下节中进一步介绍）在内的现有网络编程范例代码。这段代码大体上做了如下事情：首先构造一个数据包（包括包头和包体），然后把它发送到DNS服务器进行查询，最后接收该服务器返回的数据包并分析其结果^[注5]。

我们感兴趣的DNS数据包都是由五个独立的部分组成的：

包头

包含查询或者应答信息相关的标志位和计数器（不可省略）。

问题

包含向服务器提出的问题（出现在查询包中并且编码在响应包中）。

注5： 相关细节信息，我强烈建议你去参考RFC 1035的“Messages（消息）”那一节。

应答

包含DNS查询的应答数据（只在DNS响应包中才有）。

授权

包含说明可以从哪里获取权威响应的位置信息。

附加

包含服务器附加在查询结果后面返回的信息。

我们的程序主要关注的是前面三部分信息。我们会使用一组`pack()`命令来创建DNS包头和包体的数据结构。然后把这个数据结构交给`IO::Socket`模块发送出去。这个模块还会等待响应，并且将数据返回给我们进行解析（这时候会使用`unpack()`）。从理论上来说，这些处理并不复杂。

在我们展示代码之前，有一个细节要先讨论一下。RFC 1035^[注6]的4.1.4小节定义了DNS包中两种域名表达方式：非压缩域名和压缩域名。在非压缩表达方式中，数据包里使用的是全域名（比如`host.oog.org`），这是比较常见的。然而如果数据包中某个域名出现两次以上，那么很有可能在第二次出现时使用压缩域名格式。在压缩格式中，域信息是通过一个双字节的指针来存储的，指向第一次非压缩格式的域信息。这样就可以在一个数据包中包含`longsubdomain.longsubdomain.oog.org`域中的`host1`、`host2`和`host3`，而不必每次都包含`longsubdomain.longsubdomain.oog.org`。我们的代码支持两种表达方式，所以在其中引入了`decompress()`例程。

下面就不多说闲话了，直接看代码：

```
use IO::Socket;
use Data::Dumper;

my $hostname = $ARGV[0];
my @servers = qw(nameserver1 nameserver2 nameserver3); # 域名服务器名称
my $defdomain = '.oog.org'; # 如果没有指定就选用此默认域名

my %results;
foreach my $server (@servers) {
    $results{$server}
        = LookupAddress( $hostname, $server ); # 把查询结果存入 %results
}

my %inv = reverse %results; # 将结果哈希倒序
if ( scalar keys %inv > 1 ) { # 看看其中有多少元素
    print "There is a discrepancy between DNS servers:\n";
    print Data::Dumper->Dump( [ \%results ], ['results'] ), "\n";
}
```

注6： RFC 1035后来被RFC 1101代替了，不过这个变动不会影响我们的代码。

```

sub LookupAddress {
    my ( $hostname, $server ) = @_ ;
    my $id = 0 ;
    my ( $lformat, @labels, $count, $buf );

    ###
    ### 构造数据包头
    ###
    my $header = pack(
        'n C2 n4',
        ++$id,      # 查询编号
        1,          # qr、opcode、aa、tc、rd 等字段 (仅 rd 集合)
        0,          # ra、z、rcode
        1,          # 一个问题 (qdcnt)
        0,          # 没有应答 (ancnt)
        0,          # 授权部分没有 ns 记录 (nscount)
        0
    );              # 没有额外的 rr 记录 (arcount)

    # 如果主机名之间没有分隔符,
    # 则补上默认域名
    if ( index( $hostname, '.' ) == ?1 ) {
        $hostname .= $defdomain;
    }

    # 构造数据包 (问题中的域名) 的 qname 部分
    for ( split( /\./, $hostname ) ) {
        $lformat .= 'C a* ';
        $labels[ $count++ ] = length;
        $labels[ $count++ ] = $_;
    }

    ###
    ### 构造数据包的问题部分
    ###
    my $question = pack(
        $lformat . 'C n2',
        @labels,
        0,      # 标签结束标示
        1,      # 表示查询类型 (qtype) 为A记录
        1
    );          # 表示查询类别 (qclass) 为IN类

    ###
    ### 将数据包发送到服务器并读取响应
    ###
    my $sock = new IO::Socket::INET(
        PeerAddr => $server,
        PeerPort => 'domain',
        Proto    => 'udp'
    );

    $sock->send( $header . $question );
}

```

```

# 我们知道最大数据包的大小
$sock->recv( $buf, 512 );
close($sock);

###
### 解开数据包头部分的数据
###
my ( $id, $qr_opcode_aa_tc_rd, $ra_z_rcode, $qdcount, $ancount, $nscount,
    $arcount )
    = unpack( 'n C2 n4', $buf );

if ( !$ancount ) {
    warn "Unable to lookup data for $hostname from $server!\n";
    return;
}

###
### 解开问题部分的数据
###
# question section starts 12 bytes in
my ( $position, $qname ) = decompress( $buf, 12 );
my ( $qtype, $qclass ) = unpack( '@' . $position . 'n2', $buf );

# 快进到数据包内问题部分的结尾
$position += 4;

###
### 解开所有相关记录部分的数据
###
my ( $rtype, $rclass, $rttl, $rdlength, $rname, @results );
for ( ; $ancount; $ancount-- ) {
    ( $position, $rname ) = decompress( $buf, $position );
    ( $rtype, $rclass, $rttl, $rdlength )
        = unpack( '@' . $position . 'n2 N n', $buf );
    $position += 10;

    # 下面这行可以改用更精巧的数据结构,
    # 它目前只是简单串联所有应答
    push( @results,
        join( '.', unpack( '@' . $position . 'C' . $rdlength, $buf ) ) );
    $position += $rdlength;
}

# 对 round-robin 类型的 DNS 响应结果排序
#
# 或许应该按其语义使用定制的排序例程,
# 不过此例仅是查找每台 DNS 服务器返回的结果是否完全一致,
# 所以直接选用默认排序
return join( ', ', sort @results );
}

# 按 RFC 1035 规范处理“压缩过”的域名信息
#
# 解压缩函数根据给定的起始位置解析数据包, 并返回
# 解析完成后该域名所在的数据包位置

```

```

# （在处理完压缩格式指针之后）以及我们找到的域名
sub decompress {
    my ( $buf, $start ) = @_ ;
    my ( $domain, $i, $lenoct ) ;

    # 取得响应数据的大小，然后据此跟踪解析时刻所在的位置
    my $respsize = length($buf);

    for ( $i = $start; $i <= $respsize; ) {
        $lenoct = unpack( '@' . $i . 'C', $buf );    # 获取标签长度

        if ( !$lenoct ) {    # 0代表当前部分就此结束
            $i++;
            last;
        }

        if ( $lenoct == 192 ) {    # 如果拿到指针，递归调用
            $domain .= (
                decompress(
                    $buf, ( unpack( '@' . $i . 'n', $buf ) & 1023 )
                )
            )[1];
            $i += 2;
            last;
        }
        else {    # 否则，使用纯文本标签
            $domain .= unpack( '@' . ++$i . 'a' . $lenoct, $buf ) . '.';
            $i += $lenoct;
        }
    }
    return ( $i, $domain );
}

```

要注意这段代码和之前的范例并不等效，因为我们没有打算模拟`nslookup`行为的所有细节（比如超时、重试和搜索列表等）。在比较这三种方法的时候，请务必注意这些细节差异。

这个方法的优点：

- 这个方法完全不依赖外部程序，所以你不必了解任何外部程序的细节。
- 它可能会比调用外部程序快，起码会一样快。
- 这个方法更容易调整细节（比如超时）。

这个方法的缺陷：

- 通常写这么一大串代码要比前面的那个方式消耗更多时间并更复杂。
- 这个方法导致必须学习很多问题外的知识（即，你可能需要学习如何手动将DNS数据包组合，而这些在调用`nslookup`时并不需要了解）。

- 我们的代码并不能处理好截断的DNS应答（如果应答包太大，就会被分成多个包传送）。
- 你需要自己处理操作系统相关的问题（而这些问题往往已经被外部程序的开发人员解决了）。

使用Net::DNS

前面在第1章中说过，Perl的强大之处在于开发者的互相支持，也就是代码重用。如果你要做一件相对常见的事情，那么往往有人已经把这个问题的解决方案写成了某个模块。在我们的例子里面，可以使用Michael Fuhr所开发的卓越的Net::DNS模块来简化任务。为了使用它，我们只需要创建一个新的DNS解析对象，并且指定它使用某个DNS服务器，然后使用模块提供的某个方法来发送查询，并且用另外的方法来解析响应。

```
use Net::DNS;

my $hostname = $ARGV[0];

my @servers = qw(nameserver1 nameserver2 nameserver3 nameserver4);

my %results;
foreach my $server (@servers) {
    $results{$server}
        = LookupAddress( $hostname, $server );    # 把查询结果存入 %results
}

my %inv = reverse %results;                      # 将结果哈希倒序
if ( scalar keys %inv > 1 ) {                    # 看看其中有多少元素
    print "There is a discrepancy between DNS servers:\n";
    use Data::Dumper;
    print Data::Dumper->Dump( [ \%results ], ['results'] ), "\n";
}

# 以下代码取自 Net::DNS 的手册页，但略做修改
sub LookupAddress {
    my ( $hostname, $server ) = @_;

    my $res = new Net::DNS::Resolver;

    $res->nameservers($server);

    my $packet = $res->query($hostname);

    if ( !$packet ) {
        warn "Unable to lookup data for $hostname from $server!\n";
        return;
    }
    my (@results);
    foreach my $rr ( $packet->answer ) {
        push( @results, $rr->address );
    }
}
```

```
    return join( ' ', sort @results );  
}
```

这个方法的优点：

- 代码清晰易懂。
- 程序开发速度快。
- 依赖于模块的实现方法（纯粹用Perl实现，还是对C或C++库的包装），这段代码的运行速度应该等同于外部编译的程序。
- 这个方法的可移植性应该不错。只要模块作者已经做过这方面的努力，那么模块可以安装的环境也就是你的程序可以运行的环境。
- 如同第一个方法，编写代码更快捷，因为我们充分利用了别人的劳动成果。我们不必知道模块的实现细节，只要知道如何使用。
- 通过代码重用，我们没有重新发明轮子。

这个方法的缺陷：

- 你又回到了依赖性的老问题。你得确保模块的存在，才能保证自己的程序能够正常运行，而且必须假定模块作者的实现是正确并高效的。
- 如果模块中有bug，而且原来的作者离开了，那么你可能会成为模块的维护者。
- 可能会找不到合适的模块，或者有合适的模块但不能在你希望的操作系统上运行。

我个人喜欢用模块来完成任务。然而所有方法都可以完成任务。正所谓条条大路通罗马，所以你需要的只是开始行动！

DHCP

动态主机配置协议（DHCP）并不是一种TCP/IP名称服务，但它可以算是DNS的近亲，所以我们在这一章相继介绍这两个协议。DNS让我们了解某个主机名对应的IP地址（或IP对应的主机名），而DHCP则能让主机动态地用以太网地址来获取自己的网络配置信息（包括IP地址）。它是比BOOTP和RARP更加通用并稳定的协议，相比这两个前任，DHCP的应用场景更加广泛。

在上一节，我们并没有深入介绍DNS的工作机制^[注7]，但在这一节却没有办法忽略DHCP服

注7：例如，我们没有介绍区域传送、非ASCII主机名，甚至没有介绍单个UDP数据包不能容纳应答信息的情况下该如何处理。Paul Vixie发表在《ACM Queue》杂志2007年4月那期上的文章《DNS Complexity》深入介绍了这个主题，确实值得一读。

务器和客户机之间的交互。让我们先弄清楚这个协议的工作原理，再介绍如何使用Perl来操控。

在上一节中，我们可以说DNS客户机和服务器之间的交互模式非常简单，总是客户机提问，服务器回答。充其量也不过是客户机提问，服务器回答，然后客户机发现还需要向另外一台服务器提问才行。相比之下DHCP的交互就有趣得多，让我们来看看新的对话模式是怎样的：

DHCP客户机（广播）：有人么？我需要分配IP地址和其他配置信息。

DHCP服务器直接告诉DHCP客户机：来自某某IP地址的服务器应答。我可以让你使用某某IP地址，还有如下配置信息：……

DHCP客户机（广播）：OK，我会使用来自某某IP地址的服务器分配的某某IP地址，还有其他配置信息。

DHCP服务器告诉DHCP客户机：OK，你可以使用那个配置信息直到某某时间为止。

[时间过了一半]

DHCP客户机问服务器：Hi，我正在使用这个IP地址和某某配置信息。请问我可以继续用下去么？

DHCP服务器回答客户机：OK，没问题。你还可以使用那些信息直到某某时候为止。

[客户机准备离开网络]

DHCP客户机对服务器（可选）：OK，我已经用好IP地址和配置信息了。

这个对话可以有些变化。比如客户机可以记住自己上次的地址和租期（服务器声明的有效期），然后可以在第三步广播的时候要求再次使用。在这种情况下，服务器可以简单答复请求，然后客户机就可以接着使用老地址。另外，如果服务器或者客户机不是那么爽快，也会导致对话内容发生变化。有时候某一方可以拒绝请求，或者拒绝分配的地址，那么就必须开始新一轮的对话。要想了解所有细节，可以阅读DHCP RFC（目前编号是2131），或者参阅本章末尾列出的《*The DHCP Handbook*》。

这里明显能感觉到DHCP客户机和服务器之间的交互与之前的名称服务范例有所不同。除了必要步骤的数量差异之外，我们还能看到：

- 一个协商的握手。服务器和客户机之间需要确认对话的开始，并且就客户机将要使用的配置达成一致。双方都必须让对方接收到自己的确认信息。

- 持续状态。一旦握手发生，服务器需要在客户机的租期中跟踪地址使用情况。在达到租期一半时，客户机往往需要尝试更新协议。

注意：在这章讨论的所有其他协议中，只有WINS有比较相似的交互过程。在WINS中，客户机需要定时向服务器发送再注册请求，这样服务器可以持续获得正确的地址映射信息。DHCP相比WINS能传递更多信息，不仅仅是主机名和IP地址的映射，而且交互过程也更复杂，但它们在保持定时更新的机制上是相似的。

这两个因素导致我们在编写DHCP脚本时需要考虑更多细节。大多数情况下，这些决定会影响我们的脚本的“优雅”程度，或者说包容程度。

让我们先尝试完成两个示范任务，然后你就会明白我所说的。

主动探测不良DHCP服务器

在第13章中我们会开发一个被动不良DHCP服务器的探测程序。这里做的事情大致相似，只不过是主动方式来完成。这里主要是使用DHCP交互过程的第一步，也就是客户机发出分配地址的广播请求。所有收到了这个广播的服务器都会尝试用分配信息来响应这个请求。

注意：在我们进一步展示代码之前，先得介绍一下如何判断服务器是否“应该响应”某个请求。DHCP服务器应该是可以充分配置的。配置好的服务器可以只响应某个以太网地址的请求，或者只响应某个子网的请求，还可以按照厂商类型这样的特定标志来响应。所有不符合条件的请求应该被忽略，当然这类请求可能还是会记录在日志文件中。

我们要展示的代码是用来捕获那些配置不正确的服务器的。我们需要发现那些错误响应了（本该忽略的）请求的服务器，因为它们往往能造成大麻烦。我们应该给这些不良服务器加上正确的配置。

当然，这段代码还有些粗糙，很难捕获那些细微的配置错误，但是你应该可以逐渐改进代码，从而捕获那些非常难以识别的配置错误。

为完成这个任务，我们需要了解哪个服务器响应了我们的广播。我们几乎不检查响应的内容，当然记录响应信息也会有助于排查服务器的问题。我这样说是因为我们将要发出一些假冒的信息广播，比如假的以太网地址。

看到我们如此随便对待DHCP的请求包和响应包，你可能会意识到我们对待整个DHCP协议也不会太认真。没错，我们并不会真的和服务器协商达成租约，因为我们并非真正需要IP地址。我们感兴趣的是让尽可能多的服务器响应我们的请求。另外，如果真的尝试达成租约，可能也会导致不必要的资源分配，从而导致潜在的拒绝服务（DoS）攻击。

现在让我们来看看代码。这段代码其实并不难写，因为Stephan Hadinger的Net::DHCP::Packet模块能让我们使用面向对象的语法快速构造和解构DHCP数据包。另外我们还将使用Graham Barr的IO::Socket::INET模块来简化UDP发送和接受信息的代码（有一个小小的技巧需要额外介绍）。下面的代码应该很容易读懂：

```
use IO::Socket::INET;
use Net::DHCP::Packet;
use Net::DHCP::Constants;

my $br_addr = sockaddr_in( '67', inet_aton('255.255.255.255') );
my $xid      = int( rand(0xFFFFFFFF) );
my $chaddr   = '0016cbb7c882';

my $socket = IO::Socket::INET->new(
    Proto      => 'udp',
    Broadcast => 1,
    LocalPort => '68',
) or die "Can't create socket: $@\n";

my $discover_packet = Net::DHCP::Packet->new(
    Xid          => $xid,
    Chaddr       => $chaddr,
    Flags        => 0x8000,
    DHO_DHCP_MESSAGE_TYPE() => DHCPDISCOVER(),
    DHO_HOST_NAME()      => 'Perl Test Client',
    DHO_VENDOR_CLASS_IDENTIFIER() => 'perl',
);

$socket->send( $discover_packet->serialize(), 0, $br_addr )
    or die "Error sending: $!\n";

my $buf = '';
$socket->recv( $buf, 4096 ) or die "recvfrom() failed: $!";
my $resp = new Net::DHCP::Packet($buf);

print "Received response from: " . $socket->peerhost() . "\n";
print "Details:\n" . $resp->toString();

close($socket);
```

让我们快速浏览这段代码。开始先是定义数据包的发送目标地址（一个广播地址^[注8]），另外还有一些常量，稍后会详细介绍。然后创建发送和接受数据包所用的socket。我们为此socket设定了协议、源端口以及需要广播的特殊标志。你会发现我们并没有设置稍早定义的数据包的目标地址（也就是说这里并没有立刻使用\$br_addr）。

注8：老练的网络管理员应该能看出我们在这里使用了“全1”的广播地址，而不是针对我们的子网的广播地址。我们省略这个判断是为了让范例程序专注于其他问题。如果你真的希望精确定位广播范围，可以考虑使用其他模块，比如NetAddr::IP、Net::Interface和IO::Interface。

这是因为你不能使用`connect()`来接收一个广播响应，如同“感谢Lincole Stein，因为他使`IO::Socket`能监听广播数据包”中所介绍的那样，所以我们会在后面的某段代码中设定这个目标地址。另外你还得注意，因为使用了特殊的源地址和目标地址，我们必须使用管理员权限才能正确运行程序（例如，在Unix/Mac OS X中要通过`sudo`）。如果你不能这样运行程序，那么这段代码就会失败，报告`Permission denied`错误。

感谢Lincoln Stein，因为他使`IO::Socket`能监听广播数据包

在编写这个范例时，我几乎一整天都在为一个问题头疼。不知为什么，我不能够使用`IO::Socket::INET`来让我们在同一个socket完成广播发送和响应的接收，虽然我并不质疑它的可行性。我写过类似于Wireshark那样的监听广播数据包的代码，也写过发送广播的代码，但从来没有把两段代码如此完美地融合。

在此困境中，我做了两件事情：重新阅读了Stein那本卓越的书《*Network Programming with Perl*》(Addison-Wesley)中的相关章节；另外我还用调试器仔细跟踪了`IO::Socket::INET`模块的执行情况。在那本书中，我发现了一个关于UDP socket的`connect()`调用的段落（原书第532页）：

在数据报socket上使用连接请求，这样做的副作用是使它能接收到的数据包来源受到限制。这样做会导致其他主机或者其他端口发送的数据包被忽略……需要从多个客户机接收数据的服务器程序应该避免对端口做`connect`调用。

而我对`IO::Socket::INET`模块的跟踪也证实了这一点，在`new()`调用中使用`PeerAddr`参数设置了目标地址的端口上确实会执行`connect()`，哪怕目标地址是255.255.255.255这样的广播地址。正是这个原因导致从其他主机发送过来的数据包被忽略。

这一节展示的代码没有设置`PeerAddr`，以便监听广播数据包的响应，尽管这看上去很自然，但确实花了我不少力气。再次感谢Lincoln Stein的书帮助我在放弃之前找到了这个解决方案。

一旦有了传递数据的管道，现在需要做的就是构造数据包了。下面的内容就是构造DHCP数据包。让我们看看构造数据包时所设置的那些标志，因为这对于我们来说非常重要：

Xid

这个标志用来设定会话的事务ID。客户机在需要和服务器会话时会随机选择一个ID。这让会话的参与者能确定是在讨论同一个话题。

Chaddr

这是所谓的`client hardware address`，其实就是以太网地址或称为MAC地址。这里使用了我的笔记本的以太网地址（当然稍微调整了某些字段），因为泄露以太网地址是有风险的。更好的方法是找一块坏掉的（或者不用的）老网卡，使用它的地址，或者自动获取你的机器上的某个网卡的地址（假定它目前没有使用DHCP）。^[注9]

Flags

DHCP规范允许设定这个标志来请求响应包也以广播（而不是单播）方式发送。这里我们要求所有服务器都以广播方式响应，这样更加统一。

DHO_DHCP_MESSAGE_TYPE

这个选项用来设定要发送的DHCP数据包的类型。这里我们声明要发送的是初始的DISCOVER包。

DHO_HOST_NAME

尽管这个例子里面设定这个标志并非必须，但通过它能让客户机设定自己的识别标志。设定它的好处是让网络嗅探器更加容易追踪。在后面的某个更加复杂的范例中，我们会真的商定一个租约，设定它能让我们更加容易地搜索服务器（比如某个无线路由器）上的租约数据库。毕竟有了客户名字往往更容易识别客户身份。

DHO_VENDOR_CLASS_IDENTIFIER

这又是一个可选的标志，并非必须设置厂商类型，在DHCP的RFC里面说它能帮助简化系统配置。这个标志往往用来对网络设备进行类型区分，从而设定不同的配置信息。比如，可以让所有的Windows主机都使用Windows Server 2003 DNS服务器解析网址。如果你真的把设备进行分类，那么那些功能齐全的DHCP服务器往往就能支持这个分类。

如果说构造DHCP数据包的过程因为`Net::DHCP::Packet`模块的存在而得以简化的话，那么发送过程就更加简单了。下一步我们需要做的只是对压平的（或者说序列化之后）的数据包调用`send()`方法就可以了。这里才终于使用了之前定义的广播地址，而另外一个标志则是可选的，所以我们把它设为0，表示采用默认值。

在发送数据包之后，我们通过调用`recv()`来监听响应信息。这里有两点要注意：

注9：有件事情听上去容易做起来却不简单，那就是找到一个可移植的办法来获取主机上某个以太网卡的地址。最接近要求的方法是使用`Net::Address::Ethernet`或者`Net::Ifconfig::Wrapper`这样的模块（它们做的只不过是调用外部程序`ifconfig`或`ipconfig`），或者使用`IO::Interface::Simple`模块（它尝试使用一些系统库程序来找到信息）甚至是用`Net::SNMP::Interfaces`模块指向本地主机。但以上方法都不够完美，还不如用一个假的网卡地址来代替。

- 为了让范例代码保持简单，我们只调用一次`recv()`，这意味着我们只会收到一个响应。其实我们可以加入循环以便接收所有的响应。
- 这样简单的`recv()`调用可能会导致无限制的等待。但我们这里并没有为此牺牲代码的简洁。如果希望覆盖这个问题，请考虑使用`IO::Select`或`alarm()`方法。可以参考Lincoln Stein写的书《Network Programming with Perl》(Addison-Wesley)或者参考《Perl Cookbook》(O'Reilly)一书的例子。

为简单起见，我们假定所有意外情况都不会发生。下面的任务就是使用`Net::DHCP::Packet`分析并打印接收到的数据包。这段代码的输出如下所示：

```
Received response from: 192.168.0.1
Details:
op = BOOTREPLY
htype = HTYPE_ETHER
hlen = 6
hops = 0
xid = 912c020f
secs = 0
flags = 8000
ciaddr = 0.0.0.0
yiaddr = 192.168.0.2
siaddr = 0.0.0.0
giaddr = 0.0.0.0
chaddr = 0016cbb7c882
sname =
file =
Options :
DHO_DHCP_MESSAGE_TYPE(53) = DHCPOFFER
DHO_DHCP_SERVER_IDENTIFIER(54) = 192.168.0.1
DHO_DHCP_LEASE_TIME(51) = 86400
DHO_SUBNET_MASK(1) = 255.255.255.0
DHO_ROUTERS(3) = 192.168.0.1
DHO_DOMAIN_NAME_SERVERS(6) = 68.78.7.126 68.78.7.252
padding [270] = 0000000000000000000000000000000000000000000000000000000000000000
0000000000000000000000000000000000000000000000000000000000000000
0000000000000000000000000000000000000000000000000000000000000000
0000000000000000000000000000000000000000000000000000000000000000
0000000000000000000000000000000000000000000000000000000000000000
0000000000000000000000000000000000000000000000000000000000000000
0000000000000000000000000000000000000000000000000000000000000000
0000000000000000000000000000000000000000000000000000000000000000
```

多么美好的一个租约呀。下一步应该可以检查给我们提供租约的DHCP服务器是否就是我们期望的服务器。如果不是，那么马上开始排查错误吧！

监控正规DHCP服务器

最后我们可以把这段代码升级为另一个有用的脚本。在前面的例子中我们以

DHCPDISCOVER开始DHCP握手过程，并且收到了DHCPOFFER。但我们并没有把这个流程继续走完，所以还无法知道服务器最终能否给我们一个租约。

如果我们想知道我们的正规DHCP服务器是否能适当分配租约和配置信息呢？接下来我们必须继续下去，走完整个握手流程。

下面的范例代码能够请求一个租约，在结束之前打印这个租约并且释放掉（这是为了避免资源浪费）。我已经突出了那些有趣的地方，在代码之后加以讨论：

```
use IO::Socket::INET;
use Net::DHCP::Packet;
use Net::DHCP::Constants;

my $socket = IO::Socket::INET->new(
    Proto      => 'udp',
    Broadcast => 1,
    LocalPort => '68',
) or die "Can't create socket: $@\n";

my $br_addr = sockaddr_in( '67', inet_aton('255.255.255.255') );
my $xid      = int( rand(0xFFFFFFFF) );
my $chaddr   = '0016cbb7c882';

my $discover_packet = Net::DHCP::Packet->new(
    Xid              => $xid,
    Chaddr           => $chaddr,
    Flags            => 0x8000,
    DHO_DHCP_MESSAGE_TYPE() => DHCPDISCOVER(),
    DHO_HOST_NAME()   => 'Perl Test Client',
    DHO_VENDOR_CLASS_IDENTIFIER() => 'perl',
);

$socket->send( $discover_packet->serialize(), 0, $br_addr )
    or die "Error sending:$!\n";

my $buf = '';
$socket->recv( $buf, 4096 ) or die "recvfrom() failed:$!";
my $resp = new Net::DHCP::Packet($buf);

my $request_packet = Net::DHCP::Packet->new(
    Xid              => $xid,
    Chaddr           => $chaddr,
    DHO_DHCP_MESSAGE_TYPE() => DHCPREQUEST(),
    DHO_VENDOR_CLASS_IDENTIFIER() => 'perl',
    DHO_HOST_NAME()   => 'Perl Test Client',
    DHO_DHCP_REQUESTED_ADDRESS() => $resp->yaddr(),
    DHO_DHCP_SERVER_IDENTIFIER() =>
        $resp->getOptionValue( DHO_DHCP_SERVER_IDENTIFIER() ),
);

$socket->send( $request_packet->serialize(), 0, $br_addr )
    or die "Error sending:$!\n";
```

```

$socket->recv( $buf, 4096 ) or die "recvfrom() failed:$!";
$resp = new Net::DHCP::Packet($buf);

print $resp->toString();

my $dhcp_server = $resp->getOptionValue( DHO_DHCP_SERVER_IDENTIFIER() );

close($socket);

my $socket = IO::Socket::INET->new(
    Proto    => 'udp',
    LocalPort => '68',
    PeerPort  => '67',
    PeerAddr  => $dhcp_server,
) or die "Can't create socket: $@\n";

my $release_packet = Net::DHCP::Packet->new(
    Xid          => $xid,
    Chaddr       => $chaddr,
    DHO_DHCP_MESSAGE_TYPE() => DHCPRELEASE(),
);

$socket->send( $release_packet->serialize() )
    or die "Error sending:$!\n";

```

你可能已经大致知道了这里完成的任务，因为这是上一个例子的延续。让我们看看是哪些代码实现了更多的功能。

第一段黑体的代码显示了之前没有见过的两个标志：

DHO_DHCP_REQUESTED_ADDRESS

这用来设置我们的客户机需要服务器授予的地址。在握手过程中，我们需要把这个从服务器得到的地址重新传回去。之前DHCP通过数据包中的yiaddr字段来给我们一个建议的地址，告诉我们：“这是你可用的地址。”这里我们告诉服务器：“好的，我们就用你建议的地址。”

DHO_DHCP_SERVER_IDENTIFIER

当网络中有多个DHCP服务器的时候，它们可能都监听广播并给予响应，那么在答复时，客户机就有必要声明是针对哪个服务器的响应。这个字段就是为了设置服务器IP地址而设计的。

第二段代码的有趣之处在于使用了一个新的socket来释放租约。引入这个新的socket是因为在释放租约时不能像请求时那样使用广播，必须改用单播。而这个服务器地址可以从之前服务器响应消息中设置的DHO_DHCP_SERVER_IDENTIFIER选项获取。

最后这个改进还包含真正发送解除租约请求的代码。你可能注意到这个数据包可能是目前看到过最简单的。客户机只需要设定之前使用过的事务ID，再设定自己的以太网地址，这样就能让服务器知道到底是哪个租约需要释放。

目前我们已经展示了一种测试一个完整的DISCOVER-OFFER-REQUEST-RELEASE流程的方式。显然我们还可以进一步测试租约延续或者仔细检查服务器应答信息。如果你想继续走完租约延续的流程，那么我推荐你去看看Ming Zhang的Net::DHCPClientLive模块，它应该能节约很多时间。这个模块能发送必要的数据包，从而推动DHCP客户机在各个状态（比如“租约过期，尝试延期”）之间流转。如果你真的关心DHCP服务器的正确性，可以用这个模块来仔细测试。

本章所用模块

模块名	CPAN ID	版本
Rcs	CFRETER	1.05
Net::NIS	RIK	0.34
Data::Dumper (随 Perl 发布)	ILYAM	2.121
IO::Socket::INET (随 Perl 发布)	GBARR	1.2301
Net::DNS	OLAF	0.59
Net::DHCP::Packet	SHADINGER	0.66

更多参考资料

《DNS and BIND》（Fifth Edition），由 Paul Albitz 和 Cricket Liu 合著（O'Reilly）

《Managing NFS and NIS》（Second Edition），由 Mike Eisler et al 编著（O'Reilly）

《The DHCP Handbook》（Second Edition）由 Ralph Droms 和 Ted Lemon 合著（Sams）

《Network Programming with Perl》，由 Lincoln Stein 编著（Addison-Wesley）

《Perl Cookbook,》（Second Edition），由 Tom Christiansen 和 Nathan Torkington 合著（O'Reilly）

RFC 849: Suggestions For Improved Host Table Distribution, 作者 Mark Crispin (1983)

RFC 881: The Domain Names Plan and Schedule, 作者 J. Postel (1983)

RFC 882: Domain Names: Concepts And Facilities, 作者 P. Mockapetris (1983)

RFC 1035: Domain Names: Implementation And Specification, 作者 P. Mockapetris (1987)

RFC 1101: NS Encoding of Network Names and Other Types, 作者 P. Mockapetris (1989)

RFC 2131: Dynamic Host Configuration Protocol, 作者 R. Droms (1997)

第6章

使用配置文件工作

现在让我们来讨论平凡的配置文件。不管怎样，配置文件无处不在，不仅仅是系统管理员要和它打交道，所有软件在运行之前都需要先进行配置。没错，现今GUI程序和Web应用越来越盛行，只需点几下就能使用，但即便如此，在呈现漂亮的GUI安装界面之前，多少还是要配置一些信息的。

以Perl程序员的经验来看，典型的程序演进步骤通常都是如此。先是草草地完成最简单的脚本。比如下面这段脚本，从某个文件读入数据，遇到以hostname: 开头的行后追加域名并将数据写入另一个文件：

```
open my $DATA_FILE_H, '<', '/var/adm/data'
  or die "unable to open datafile: $!\n";
open my $OUTPUT_FILE_H, '>', '/var/adm/output'
  or die "unable to write to outfile: $!\n";

while ( my $dataline = <$DATA_FILE_H> ) {
  chomp($dataline);
  if ( $dataline =~ /^hostname: / ) {
    $dataline .= '.example.edu';
  }
  print $OUTPUT_FILE_H $dataline . "\n";
}
close $DATA_FILE_H;
close $OUTPUT_FILE_H;
```

然后程序很快就会改成下面这样，提取会用到的参数，集中安置到程序顶部作为配置段：

```
my $datafile    = '/var/adm/data';    # 输入数据文件名
my $outfile     = '/var/adm/output';  # 输出数据文件名
my $change_tag  = 'hostname: ';      # 要追加数据的行首内容
my $fdqn        = '.example.edu';    # 将要追加的域名

open my $DATA_FILE_H, '<', $datafile
  or die "unable to open $datafile: $!\n";
open my $OUTPUT_FILE_H, '>', $outfile
  or die "unable to write to $outfile: $!\n";
```



```

while ( my $dataline = <$DATA_FILE_H> ) {
    chomp($dataline);
    if ( $dataline =~ /^$change_tag/ ) {
        $dataline .= $fdqn;
    }
    print $OUTPUT_FILE_H $dataline . "\n";
}
close $DATA_FILE_H;
close $OUTPUT_FILE_H;

```

许多Perl程序就此打住不再变化，在以后的日子里一直保持这种配置参数的方式。不过，经验老道的程序员明白其中潜在的风险：随着开发过程的持续推进，程序会变得越来越大，各种问题也会随之涌现。若是再把程序交付给其他人维护的话，更是如此。

这种问题最初都是因为在程序内部比较深入的地方增加代码，不小心篡改了参数\$change_tag或\$fdqn的值所造成。于是顷刻间，程序输出的内容变得面目全非，无可复加。这里列出的程序非常简单，要发现问题自然也很容易，什么地方修改了\$change_tag或\$fdqn几乎可以一目了然，但实践中使用的程序可能极为庞杂，源代码就要翻上好几屏，要一眼洞穿问题所在谈何容易。

将\$fdqn变量名字改得更直白一些，比如\$dont_change_this_value_yesiree_bob，貌似可以解决这种问题，但这不是什么好主意。抛开每次耗费如此多按键输入不谈，这么做还降低了程序的可读性。类似的数据隐藏技巧还有许多（如使用闭包、符号表操作等等），但都无助于改善代码的可读性，反而会使程序变得无谓复杂。

最好的办法是采纳与use constants编译指令（见第189页的“常量，恰如北极星”）类似的风格，将参数改为只读变量：

```

use Readonly;

# 将配置参数变量名改为全大写的，以突出显示
# 注意：以下为 Perl 5.8.x 语法。早先版本请查阅 Readonly 文档
Readonly my $DATAFILE => '/var/adm/data'; # 输入数据文件名
Readonly my $OUTPUTFILE => '/var/adm/output'; # 输出数据文件名

Readonly my $CHANGE_TAG => 'hostname: '; # 要追加数据的行首内容
Readonly my $FDQN => '.example.edu'; # 将要追加的域名

open my $DATA_FILE_H, '<', $DATAFILE
    or die "unable to open $DATAFILE: $!\n";
open my $OUTPUT_FILE_H, '>', $OUTPUTFILE
    or die "unable to write to $OUTPUTFILE: $!\n";

while ( my $dataline = <$DATA_FILE_H> ) {
    chomp($dataline);
    if ( $dataline =~ /^$CHANGE_TAG/ ) {
        $dataline .= $FDQN;
    }
}

```

```
}
print $OUTPUT_FILE $dataline . "\n";
}
close $DATA_FILE_H;
close $OUTPUT_FILE_H;
```

常量，恰如北极星

为什么不直接用“use constants”代替呢？Readonly 模块在其文档中给出了多条理由，相比之下最重要的是以下三条：

1. 在字符串中内插Readonly变量的能力（如`print "Constant set to $CONSTANT\n"`）。
2. 在词法作用域内使用Readonly变量的能力（如`Readonly my $constant => "fred"`）。所以可根据自己的需要，仅在小范围内使用只读变量。
3. 和`use constant`不同，一旦Readonly变量定义好，重新定义会被断然拒绝。

现在，所有的配置参数都集中到了脚本顶部，^[注1]接下来会面临的一个问题是：继续编写第二个、第三个使用类似参数的脚本该怎么办？显然，复制粘贴不是正确答案。简单地复制看似无关痛痒，但这会导致多源真理（Multiple Sources of Truth?）。^[注2]哪天应需而变时，会很容易忘记更新其他脚本中的配置参数，然后再花上几个小时诊断问题缘由。这还是刚开始，要是破坏了原来的数据，岂不是更糟？所以，请不要这么做。

正确的做法是另外创建单独的配置文件（或是其他配置策略，本章稍后会谈及）。接下来要决定的是配置文件该选取何种格式。

怎么说呢，答案正如一句老笑话，“规范和准则的美妙之处，恰恰在于可以有那么多的规范和准则可以选用！”^[译注]哪种格式才是最好的，取决于应用场景、开发策略以及个人喜好等诸多因素。我个人是个多元主义者，所以接下来我会介绍几种最常见的配置文件格式，究竟用哪种由你自己来决定。

配置文件格式

本章我们会讨论四种配置文件格式：二进制格式、使用分隔符的文本格式、键-值对格

注1： 有时我们也会利用 `__DATA__` 存储数据，不过一般来讲，参数置顶才是更好的选择，且便于编辑和查看。

注2： 这里引用 Hugh Brown 的话借喻。

译注：原话为 “The wonderful thing about standards is there are so many to choose from!”

式以及使用置标语言的XML和YAML格式。每种格式我都会提供一些个人参考意见，你可以权衡使用。

二进制格式

第一种配置文件格式是我最不爱用的，先来说掉算了。早期许多程序员都喜欢直接把内存中的Perl数据结构序列化到文件中存储，当作保存配置数据的一种方式。有好多序列化数据的方法，用得最早最多的还是Storable模块：

```
use Storable;

# 将配置文件数据结构写出到 $CONFIG_FILE
store \%config, $CONFIG_FILE; # 可改用 nstore() 保存为跨平台文件

# 在程序其他部分（或者别的程序里），从该文件读入配置信息
my $config = retrieve($CONFIG_FILE);
```

该模块是用C语言实现的，如果改用纯Perl实现的话，DBM::Deep模块倒是另一种比较好的选择。它默认保存的文件格式就是跨平台的（否则就得用Storable的nstore()方法）：

```
use DBM::Deep;

my $configdb = new DBM::Deep 'config.db';

# 将主机配置信息存入该数据库
$configdb->{hosts} = {
    'agatha'    => '192.168.0.4',
    'gilgamesh' => '192.168.0.5',
    'tarsus'    => '192.168.0.6',
};

# （其后）提取存储在数据库中的所有主机名
print join( ' ', keys %{ $configdb->{hosts} } ) . "\n";
```

二进制文件中的数据读取可以相当快，所以如果在乎性能的话，可以考虑采用这种格式。此外，将原始数据内容（比如内存中复杂的Perl数据结构）原样封存到二进制文件中，不仅免去了数据格式间转换的繁琐过程，还延续了数据的生命期，非常方便。

那为什么我不太喜欢这种配置文件呢？对我来讲，最令我不爽的是文件内容的封闭和隐晦。我希望可以在任何时候都能方便查看当前配置，所以保存的内容应该可以直接阅读而不用借助任何解析程序（同样，更新配置信息时也是如此）。另外，二进制格式的数据无法用标准的外部工具处理，像常见的grep等。^[注3]当然，选择并非唯一的。如果确实在乎读写速度，还是有些别的格式可以使用，稍后就会看到。

注3：正如 Mark Pilgrim 曾经说过的，“永远不要相信无法用 Emacs或vi编辑的文件格式（Never trust a format you can't edit in Emacs or vi）”（<http://dashes.com/anil/2003/11/tools-affect-co.html#comment-2725>）。

使用分隔符的文本格式

按照特定字符将数据分隔成不同列表示的文件格式，同样是我不太喜欢的一种格式。Unix系统上/etc目录中好多配置文件都喜欢用这种格式：*passwd*、*group*等等。用逗号或特定字符分隔数据的文件格式（CSV格式，指Comma-Separated或Character-Separated Value）同属此类。

在Perl里面读取这种格式的数据非常方便，直接利用内置函数`split()`就可以：

```
use Readonly;

Readonly my $DELIMITER => ':';
Readonly my $NUMFIELDS => 4;

# 此处打开配置文件，读入一行，存到 $line_of_config

# 解析各字段数据
my ( $field1, $field2, $field3, $field4, $excess ) =
    split $DELIMITER, $line_of_config, $NUMFIELDS;
```

对于CSV文件，则可以用专属的解析模块来处理像转义字符（比如用逗号作为分隔符时，数据字段中出现的逗号则需要转义）这样的棘手情况。`Text::CSV::Simple`是围绕`Text::CSV_XS`的包装，用起来非常方便：

```
use Text::CSV::Simple;

my $csv_parser = Text::CSV::Simple->new;

# 返回的@data为二级数组，一级元素表示行，二级元素表示各个字段内容
my @data = $csv_parser->read_file($datafile);
```

我不喜欢这种格式也是有原因的。虽然它比二进制格式好些，可以为人所读，也能用外部工具搜索处理，不过对人来说仍旧不够友好。如果没有好记性或外部文档说明，几乎很难理解或弄清每个字段的意义（“第7列表示什么意思？”），而且输入不够细心的话还很容易弄乱字段数据。不仅如此，字段顺序也容易颠倒混淆。如果你还是坚持选用这种格式，那么随着工作的展开，你会发现各种应用程序都有自己的一套处理字段内容中逗号、引号、回车符的办法。表面上看CSV格式易于操作，但实际仍需借助特定工具生成或解读。

键-值对格式

最常见的键-值对格式采用`key{something}value`写法，其中的`{something}`一般为空白字符、冒号或等号。某些键-值对格式（比如.ini文件）还有一些扩展变化，比如增加分组小节名称：

```
[server section]
{setting}={value}
{setting}={value}
```

或者配置作用域（像Apache的配置文件中）：

```
<Location>
    {setting} {value}
    {setting} {value}
    ...
</Location>
```

初次用Perl处理键-值对格式文件貌似有点困难，因为可供选用的模块实在太多了，在写本书时CPAN中就有至少26种。

该选择哪个模块呢？可以先问自己一些问题，然后逐步确定需求，继而缩小选择范围。

作为开始，首先要问自己配置文件会有多复杂：

最简单的.ini文件是否够用？还是需要更复杂的.ini？Apache风格？还是扩展的Apache风格？需要用到分组小节吗？需要在作用域内配置参数或指令吗？想要实现自己的格式文法吗？

接下来，考虑如何和配置信息交互：

想要模块返回简单的数据结构，还是按对象来表示信息？用tie指令绑定到哈希来操作数据，还是用普通的Perl常量的形式？保存配置信息时是否需要按照原来读取时的顺序？能自动识别文件格式的模块会有帮助吗？

最后，想想还有哪些方面对你而言不可或缺：

是否在乎解析速度，或者解析时消耗的内存？重载配置时是否需要缓存机制？是否需要将配置信息级联到一起（即在全局配置文件中包含其他配置文件）？在解析配置信息时是否需要验证数据的有效性？是否需要模块同时支持读写功能？

每个问题的回答都会最终引导你选用合适的模块。限于篇幅，我不可能一一介绍所有模块，所以下面选择了三个比较有代表性的模块。

Config::Std是Damian Conway写的配置文件解析模块。它在更新配置文件时，会保留原来的分组顺序以及相应的注释。它所处理的配置文件格式和.ini文件颇为相近，初次使用也很容易理解。下面是该模块的大致用法（好像本节的代码范例都非常简单，可能读起来有些乏味，不过这些模块本就是为简化使用而设计的，用法自然直白）：

```
use Config::Std;
```

```

read_config 'config.cfg' => my %config;

# 在代码中使用或修改参数 $config{Section}{key}...
...
# 写回配置文件
write_config %config;

```

Config::Std模块比较简单，如果碰到它无法应付的情况，《Conway在他的Perl Best Practices》（O'Reilly）一书中建议改用Thomas Linden写的更强劲的Config::General模块。它不仅支持Apache系列配置风格，语法也更丰富灵活。该模块的用法并不比Config::Std复杂：

```

use Config::General;

my %config = ParseConfig( -ConfigFile => 'rcfile' );

# 在代码中使用或修改参数 %config...
...
# 写回配置文件
SaveConfig( 'configdb', \%config );

```

如果Config::General还不能满足需要，试试Karl Gaissmaier的Config::Scoped模块。该模块支持各种复杂的配置格式，包括按作用域定义的形式（如BIND或ISC DHCP服务器所用的配置），可以在解析数据时做合法性校验，同时还能检查配置文件的读写权限。此外，它还支持数据缓存功能，可以在首次读入大量配置信息后，以二进制形式保存解析得到的数据结构，以后只要原始文件未更新，就直接读取二进制镜像。这种做法结合了二进制文件的高性能和文本文件的可读性这两个优点。但它无法在程序中动态更新配置内容，这一点不如上面介绍过的那些模块方便。下面的代码片段演示了如何使用缓存功能：

```

use Config::Scoped;
my $parser = Config::Scoped->new( file => 'config.cfg' );
my $config = $parser->parse;

# 将缓存后的配置信息存入二进制文件，以备后用
$parser->store_cache( cache => 'config.cfg.cache' );

# （之后，或在另一程序中，加载缓存版本）
my $cfg = Config::Scoped->new( file => 'config.cfg' )->retrieve_cache;

```

如果你喜欢自己动手实现，仍有一堆模块可供使用，如David Schweikert的Config::Grammar模块，它允许自定义配置文件的格式文法。我个人不建议创造私有格式，这样多少会降低可维护性，不过确实需要的话，Schweikert的模块还是能帮上忙的。

置标语言

之前介绍的每一种配置文件格式都有这样或那样的缺憾，要么缺乏可读性，要么缺乏可扩展性，并且解析数据时在不同程度上都有着不可预测性：数据规格缺少精细控制，格式死板不够灵活，无法验证解析后的数据是否正确等。比如CSV文件字段内出现的分隔符，不同的解析程序处理方式各不相同。在二进制文件中，只要unpack()模板中的一个元素弄错，读出来的便全是乱码一样的无用数据，如果不查看结果根本无法知道解析程序出错了。而置标语言，善用活用，则无往而不胜。

XML

在我开始编写本书第一版时，XML是一门刚刚起步并蓬勃发展的新技术，广泛受到系统管理员的青睐。这里正好说到有关XML的应用，也就顺便作些介绍。^[注4]过去几年间，在系统管理和数据处理的各个方面，都可以看到XML的用武之地，作为基础设施的紧密组成部分，XML几乎不可或缺。既然XML如此重要，我们不妨花点时间看看相关概念和知识。

XML开始变得流行起来的地方之一就是配置文件中。事实上，XML还有诸多派生出来的专用方言（此处我们不展开讨论），包括DCML、NetML和SAML等等，它们有希望成为各个管理领域中配置文件格式的实际标准。

在我们深入探讨使用XML之前，先来看看为什么它适合用来配置信息。XML本身固有的一些属性让它成为配置文件的好选择：

- XML是一种纯文本格式，也就是说我们使用Perl工具包可以很容易地编辑或处理它。
- 保持简单（因为复杂晦涩的XML文档和其他格式一样令人难以捉摸）时，XML还是具有一定的自我描述性的，实际上相当于一份自我说明文档。如字符分隔文件/*etc/passwd*，总是令人难以确定各个部分代表什么。而在XML里永远不存在这样的问题，每个字段内容都由特定标签环绕来说明其具体意义。
- XML具备一定的自我验证能力。如果使用验证解析器，一个条目的格式中的错误会被立该发现，因为文件将无法根据其文档类型定义（Document Type Definition，DTD）或模式（schema）来正确解析。就算没有验证解析器，一般的XML工具也会对基础的语法格式进行检查，并能找出大多数错误。^[注5]

注4：本书第一版中，关于XML的讨论放在用户账户管理维护的相关章节中，不过这些年来，在系统管理领域XML渐已成为语义化数据的存储交换格式，所以移到本章介绍更为妥当。

注5：最容易出现的错误是数据截断。如果遗漏结束标签，后续的XML内容都会被当作该标签内的数据，最后发现标签不配对，自然无法通过格式验证。但我们稍后要介绍的YAML就不存在这种问题。

- XML在描述数据时也很灵活，不管是标签的命名还是数据的层次结构，你都可以按照最自然的方式定义。所以不必为每种格式都写一款解析工具，只要一种通用的解析工具就能访问所有数据。

下面的例子虽然简单，但基本上可以说明XML具备自我描述能力。无需另外的手册页，我们都能明白这段XML所表示的意义：

```
<network>
  <host>
    <name>agatha</name>
    <addr>192.168.0.4</addr>
  </host>
  ...
</network>
```

我们接下来要用的配置文件与此类似，但不会这么简单，你很快就会看到。

用Perl输出XML文件

既然XML是文本文件格式，要在Perl里输出XML文件并非什么难事。最简单的方法是直接用print语句输出所有符合XML格式的字符，但我们不会这么做，因为有更好的办法。用Benjamin Holzman的XML::Generator模块，或者David Megginson的XML::Writer模块，这样操作过程更简单，还能避免手动处理标签时容易发生遗漏的问题，对于标签中的数据也会自动进行转义（如<、>、&等特殊字符）。

为使代码简洁，我们就试着输出上面例子中的XML配置文件。^[注6]模块XML::Writer的用法如下：

```
use XML::Writer;
use IO::File;

my %hosts = (
    'name' => 'agatha',
    'addr' => '192.168.0.4',
);

my $FH = new IO::File('>netconfig.xml')
    or die "Unable to write to file netconfig.xml: $!\n";

my $xmlw = new XML::Writer( OUTPUT => $FH );

$xmlw->startTag('network');
print $FH "\n ";
```

注6：快速提示：XML 规范建议每个 XML 文件最好在开始时声明版本（比如 `<?xml version="1.0"?>`）。这不是强制的，但如果要遵循，可以用 XML::Writer 提供的 `xmlDecl()` 方法自动生成。


```

$xmlw->startTag('host');

# 注意，这里的代码没有限定输出顺序
foreach my $field ( keys %hosts ) {
    print $FH "\n      ";
    $xmlw->startTag($field);
    $xmlw->characters( $hosts{$field} );
    $xmlw->endTag;
}
print $FH "\n ";

$xmlw->endTag;
print $FH "\n";

$xmlw->endTag;
$xmlw->end;
$FH->close();

```

用XML::Writer模块还有许多额外好处：

- 代码本身非常清晰易懂，任何对置标语言有所接触的人看一眼这些方法的名字startTag()、characters()和endTag()，都能推测出个大概来。
- 尽管我们这里的数据无需转义，不过用characters()处理一遍总是好的，它会帮我们替换掉可能产生歧义的字符（比如>）。
- 写代码时不必记住最近打开的标签名字，直接用XML::Writer的endTag()方法关闭标签就行。

但就本例而言，使用XML::Writer也有些弊端：

- 键入的字符显然比直接用print时要多。
- XML::Writer生成的XML文件更适合于机器分析，而不是供人阅读。它是有许多初始化时用到的选项以方便输出更为齐整的内容，不过仍然无法得到上面例子中那样的结果。而用print \$FH的话，控制好缩进空格就能得到漂亮的输出。

解析与操作XML的最佳实践工具概览

接下来我们看看在Perl里解析XML的各种方法。每种方法都各有所长，适合于不同的应用场景。全部了解后，可以在实践中选择最适合的一种。

为便于比较和理解各种解析工具，我们选用同一份范例XML文件作为输入。先来看一下它的内容，好让我们对其表达的数据大致有个了解，然后再作解析处理。以下是完整的XML文件，稍后我们会逐段讲解：

```
<?xml version="1.0" encoding="UTF-8"?>
```

```

<network>
  <description name="Boston">
    This is the configuration of our network in the Boston office.
  </description>
  <host name="agatha" type="server" os="linux">
    <interface name="eth0" type="Ethernet" >
      <arec>agatha.example.edu</arec>
      <cname>mail.example.edu</cname>
      <addr>192.168.0.4</addr>
    </interface>
    <service>SMTP</service>
    <service>POP3</service>
    <service>IMAP4</service>
  </host>
  <host name="gil" type="server" os="linux">
    <interface name="eth0" type="Ethernet" >
      <arec>gil.example.edu</arec>
      <cname>www.example.edu</cname>
      <addr>192.168.0.5</addr>
    </interface>
    <service>HTTP</service>
    <service>HTTPS</service>
  </host>
  <host name="baron" type="server" os="linux">
    <interface name="eth0" type="Ethernet" >
      <arec>baron.example.edu</arec>
      <cname>dns.example.edu</cname>
      <cname>ntp.example.edu</cname>
      <cname>ldap.example.edu</cname>
      <addr>192.168.0.6</addr>
    </interface>
    <service>DNS</service>
    <service>NTP</service>
    <service>LDAP</service>
    <service>LDAPS</service>
  </host>
  <host name="mr-tock" type="server" os="openbsd">
    <interface name="fxp0" type="Ethernet">
      <arec>mr-tock.example.edu</arec>
      <cname>fw.example.edu</cname>
      <addr>192.168.0.1</addr>
    </interface>
    <service>firewall</service>
  </host>
  <host name="krosp" type="client" os="osx">
    <interface name="en0" type="Ethernet" >
      <arec>krosp.example.edu</arec>
      <addr>192.168.0.100</addr>
    </interface>
    <interface name="en1" type="AirPort">
      <arec>krosp.wireless.example.edu</arec>
      <addr>192.168.100.100</addr>
    </interface>
  </host>
  <host name="zeetha" type="client" os="osx">

```

```

    <interface name="en0" type="Ethernet" >
      <arec>zeetha.example.edu</arec>
      <addr>192.168.0.101</addr>
    </interface>
    <interface name="en1" type="AirPort">
      <arec>zeetha.wireless.example.edu</arec>
      <addr>192.168.100.101</addr>
    </interface>
  </host>
</network>

```

该文件描述了一个小型网络，其中有三台服务器和两台终端机。每个<host></host>元素都代表着一台机器，第一台是提供邮件服务的服务器：

```

<host name="agatha" type="server" os="linux">
  <interface name="eth0" type="Ethernet" >
    <arec>agatha.example.edu</arec>
    <cname>mail.example.edu</cname>
    <addr>192.168.0.4</addr>
  </interface>
  <service>SMTP</service>
  <service>POP3</service>
  <service>IMAP4</service>
</host>

```

每个网络接口都有一个<arec></arec>表示该主机的DNS域名（用DNS的话来讲，是一条A记录）。该服务器提供了三种与邮件相关的服务，并且都可以通过DNS CNAME元素表示的域名访问。另外两台服务器提供其他服务，分别对应于不同的CNAME记录。下面是终端机的配置：

```

<host name="krosp" type="client" os="osx">
  <interface name="en0" type="Ethernet" >
    <arec>krosp.example.edu</arec>
    <addr>192.168.0.100</addr>
  </interface>
  <interface name="en1" type="AirPort">
    <arec>krosp.wireless.example.edu</arec>
    <addr>192.168.100.100</addr>
  </interface>
</host>

```

这里的配置和服务器不同，除了type属性不一样且无相关服务外，还有多个网络接口（应该是一台笔记本，因为有无线的AirPort和有线的以太网接口），可以通过不同线路连接到服务器。每个接口都分配有一个独立IP地址和DNS主机名（A记录）。

该配置文件的写法和之前的XML例子有些不同，我们可以看到分别使用了属性（name、type、os）和子元素（interface、service）两种写法。到底什么时候该用属性什么时候该用子元素，历来都是仁者见仁，智者见智。我是参考了关于此主题的经典讨论

(<http://xml.coverpages.org/attrSperberg92.html>), 把描述机器的客观信息写成XML属性, 其他可能会增减变化的(网络接口、提供的服务等)东西写成子元素。当然这只是我的个人意见, 你要怎么写随你, 只要对你来讲说得通就行。

那么, 我们开始吧, 接下来我会由浅入深介绍三种不同的模块/实现方法。每种模块都各有优缺点, 这要看应用场景而定, 我会把这些优缺点放在每个小节开始的地方说明, 这样你就会有大概的把握和预期。全部看完后应该会有粗浅的认识, 以后需要用哪种办法也就有方向了。^[注7]

用XML::Simple解析

XML::Simple的优缺点

优点是:

- 这是用法最简单的模块(至少从表面上来看)。
- 它会把XML转化为对应的Perl数据结构。
- 为求快速, 它实际上是调用了别的模块(XML::LibXML)负责实际的解析, 稍后我们会介绍。
- 文档完备。
- 模块维护者尽职尽责, 积极回应提问及错误报告。

缺点是:

- 得到的元素不会保留原始XML中的顺序和格式(也就是说, 再次读入XML文件并解析, 返回的数据顺序和格式可能有所不同)。
- 无法处理“混杂内容”的情形(元素中包含其他元素和文本), 像这样:

```
<network>This is our <type>devel</type> network</network>
```
- 默认会把所有数据一次读入内存。
- 有些人反对直接将XML转换成Perl数据结构, 因为逻辑上两者并不等价对应。

那么, 何时该用此模块? 对于像XML配置文件这样的小型XML相关作用来说, 用XML::Simple就已足够。如果目的就是读取XML文件或编写XML文件, 用它没问

注7: 如果要自己编写代码来解析XML文件当然也是可以的(我想你大概是想用正则表达式直接提取吧)。不过, 这可是吃力不讨好的事情, 所花的时间更多, 得到的结果却未必正确。一定要用正则表达式来实现超级简单的XML解析工具, 倒是有些现成的模块可用, 不过我们此处从略。

题（要是先读取再写回去，得注意数据保存顺序，这要看应用程序是否对此有要求了）。就我而言，如果XML文件仅是实际任务中用于配置的一小部分而非业务重点的话，我就会用这个模块。

最简单的方法就是用Grant McLean的XML::Simple模块读取XML配置文件。可以用寥寥几句话，把XML文件一次读入并转成Perl数据结构：

```
use XML::Simple;

my $config = XMLin('config.xml');

# 接下来可以访问配置参数 $config->{stuff}
```

修改数据内容后写回XML配置文件一样简单轻松：

```
... （此时数据结构已经解析载入）

XMLout($config, OutputFile => $configfile );
```

我知道你在想什么：不会真的就这么简单吧？好吧，我们来看看幕后究竟发生了什么。打开Perl自带的调试器，运行上面第一段XML::Simple代码^[译注2]，然后用x \$config命令打印数据结构，会看到下面这段输出：^[注8]

```
0 HASH(0xa36fc8)
  'description' => HASH(0x97f5e8)
    'content' => '
      This is the configuration of our network in the Boston office.
    '
  'name' => 'Boston'
  'host' => HASH(0xa5d6ac)
    'agatha' => HASH(0xa5d424)
      'interface' => HASH(0xa38f98)
        'addr' => '192.168.0.4'
        'arec' => 'agatha.example.edu'
        'cname' => 'mail.example.edu'
        'name' => 'eth0'
        'type' => 'Ethernet'
      'os' => 'linux'
      'service' => ARRAY(0xa5da6c)
        0 'SMTP'
        1 'POP3'
        2 'IMAP4'
      'type' => 'server'
    'baron' => HASH(0xa51390)
      'interface' => HASH(0xa3e228)
```

注8： 调试 XML::Simple 代码，最好选用专门的数据结构打印模块，比如 Data::Dumper、Data::Dump::Streamer、YAML 或用 Perl 自带的调试工具。

译注2：把这两行保存为read.pl脚本，然后运行perl -dread.pl，再键入r运行程序。

```

        'addr' => '192.168.0.6'
        'arec' => 'baron.example.edu'
        'cname' => ARRAY(0xa5d874)
            0 'dns.example.edu'
            1 'ntp.example.edu'
            2 'ldap.example.edu'
        'name' => 'eth0'
        'type' => 'Ethernet'
        'os' => 'linux'
        'service' => ARRAY(0xa5d994)
            0 'DNS'
            1 'NTP'
            2 'LDAP'
            3 'LDAPS'
        'type' => 'server'
    'gil' => HASH(0xa5d61c)
        'interface' => HASH(0xa3de44)
            'addr' => '192.168.0.5'
            'arec' => 'gil.example.edu'
            'cname' => 'www.example.edu'
            'name' => 'eth0'
            'type' => 'Ethernet'
            'os' => 'linux'
            'service' => ARRAY(0xa5d964)
                0 'HTTP'
                1 'HTTPS'
            'type' => 'server'
    'krosp' => HASH(0xa5d754)
        'interface' => HASH(0xa5d664)
            'en0' => HASH(0xa5d5ec)
                'addr' => '192.168.0.100'
                'arec' => 'krosp.example.edu'
                'type' => 'Ethernet'
            'en1' => HASH(0xa5d604)
                'addr' => '192.168.100.100'
                'arec' => 'krosp.wireless.example.edu'
                'type' => 'AirPort'
            'os' => 'osx'
            'type' => 'client'
    'mr-tock' => HASH(0xa4ee28)
        'interface' => HASH(0xa2fc50)
            'addr' => '192.168.0.1'
            'arec' => 'mr-tock.example.edu'
            'cname' => 'fw.example.edu'
            'name' => 'fxp0'
            'type' => 'Ethernet'
            'os' => 'openbsd'
            'service' => 'firewall'
            'type' => 'server'
    'zeetha' => HASH(0xa5d4b4)
        'interface' => HASH(0xa5d4cc)
            'en0' => HASH(0xa5d454)
                'addr' => '192.168.0.101'
                'arec' => 'zeetha.example.edu'
                'type' => 'Ethernet'

```

```

'en1' => HASH(0xa5d46c)
'addr' => '192.168.100.101'
'arec' => 'zeetha.wireless.example.edu'
'type' => 'AirPort'
'os' => 'osx'
'type' => 'client'

```

仅是按照默认用法，我们就得到了一个可供使用的数据结构。返回的哈希有一个`host`键，对应值为另一个以所有主机名为键的哈希。也就是说，我们可以直接通过 `$config->{host}->{hostname}` 获取该主机所有相关信息。很好很强大，不过要是仔细观察这个数据结构，会发现事实并非如此，还有几个问题要解决（以下按照重要程度由低到高分别讨论）：

1. `XML::Simple` 会保留原始数据中的所有空格，包括那些加在数据首尾以便在XML中看起来格式漂亮整齐的空格。实际操作往往不需要这些首尾空格，自己写代码删除这些空格当然可行，只不过每个地方都要处理，还不如直接让解析器来帮我们去掉。把原来的：

```
my $config = XMLin('config.xml');
```

改为：

```
my $config = XMLin('config.xml', NormalizeSpace => 2);
```

就可以达成。

2. 如果仔细观察所有服务器的`<service></service>`元素是如何转化为Perl数据结构的，你会发现这里有个陷阱。看看`agatha`和`mr-tock`的结果好了。`agatha`的服务有三个：

```

'service' => ARRAY(0xa4f2d8)
  0  'SMTP'
  1  'POP3'
  2  'IMAP4'

```

而 `mr-tock` 的服务却只有一个：

```
'service' => 'firewall'
```

前者按照数组返回，后者直接选用标量。这种差异让代码编写变得十分困难，必须先判断返回的数据类型然后分别提取处理。为什么对于同样的元素`XML::Simple`的做法却不同呢？因为默认`XML::Simple`会根据嵌套元素（在`<service></service>`里面的内容）出现的个数决定返回数据列表还是单项数据。我们可以用`ForceArray`参数指定强制使用数组返回。如果参数值为1，则所有嵌套元素均以数组形式返回，但也可以具体指定某些元素以数组返回，比如改写成：

```
my $config = XMLin('config.xml', NormalizeSpace => 2, ForceArray => ['service']);
```

之前两段主机的服务列表便会按照统一的数据结构返回：

```
'service' => ARRAY(0xa4f2d8)
    0  'SMTP'
    1  'POP3'
    2  'IMAP4'
...
'service' => ARRAY(0xa31fac)
    0  'firewall'
```

何时该用XML::Simple的ForceArray选项

你可能会说，总是用ForceArray => 1不就好了吗，代码也统一，不用多考虑什么。没错，不过这也要看情况。XML 文件中都是单个内容的元素，你很快就会厌倦每次都写数组下标取用第一个也是唯一一个元素。

我建议按照这样的思路判断使用ForceArray的方式：从逻辑上讲，如果某个元素应该或有可能包含多个子元素实例时（比如多重 <service></service>子元素），则在 ForceArray列表中指定。如果该元素只可能^[注9]有一个子元素实例，那就留着不要动它。

此外，如果打算用KeyAttr选项（稍后就会介绍），列在该选项中的元素也必须一并列在ForceArray列表中。

3. 还有个XMLin()返回得不一致的地方。仔细观察所有主机的<interface></interface>元素是怎样转化成Perl数据结构的，比较agatha和zeetha的结果就能说明问题。agatha的接口描述如下：

```
'interface' => HASH(0xa38f98)
  'addr' => '192.168.0.4'
  'arec' => 'agatha.example.edu'
  'cname' => 'mail.example.edu'
  'name' => 'eth0'
  'type' => 'Ethernet'
```

而zeetha的接口描述却是另一番模样：

```
'interface' => HASH(0xa5d4cc)
  'en0' => HASH(0xa5d454)
    'addr' => '192.168.0.101'
    'arec' => 'zeetha.example.edu'
    'type' => 'Ethernet'
```

注9：有经验的老手看到这里没准会偷笑。说到“只可能”或“绝对”，只要有 DTD 或者相应的 XML 模式(schema)定义，我们总是可以说“绝对”这个词，否则就是用户给出的数据有问题。所以为了避免实际数据和预期不一致，应该先对 XML 作有效性验证。


```

'en1' => HASH(0xa5d46c)
'addr' => '192.168.100.101'
'arec' => 'zeetha.wireless.example.edu'
'type' => 'AirPort'

```

单个接口时，返回的是其键为组成该接口的各个组件的哈希（接口的地址、类型、域名等等）；而有多接口时，返回的是哈希的哈希，它以各接口名为键，以描述该接口具体信息的哈希为值。像这种情况，我们又不得不分两套代码，先判断数据结构的形式，而后再取用原本相同意义的数据。

这里又遇到了和之前类似的情形，相同子元素（`<interface></interface>`）出现单一实例和多个实例时返回的数据结构不一致。试试使用`ForceArray`选项进行强制转换：

```

my $config = XMLin('config.xml', NormalizeSpace => 2,
                    ForceArray      => ['interface']);

```

太棒了，现在两者看起来整齐一致：

```

'interface' => HASH(0xa32f28)
  'eth0' => HASH(0xa32e80)
    'addr' => '192.168.0.4'
    'arec' => 'agatha.example.edu'
    'cname' => 'mail.example.edu'
    'type' => 'Ethernet'
...
'interface' => HASH(0xa2593c)
  'en0' => HASH(0xa257bc)
    'addr' => '192.168.0.101'
    'arec' => 'zeetha.example.edu'
    'type' => 'Ethernet'
  'en1' => HASH(0xa257d4)
    'addr' => '192.168.100.101'
    'arec' => 'zeetha.wireless.example.edu'
    'type' => 'AirPort'

```

且慢，我们用的不是`ForceArray`吗？难道强制数组也能返回哈希形式的列表？显然`<interface></interface>`元素被转成了哈希的哈希，它根本不是严格意义上的数组。这正是`XMLin()`的神奇之处，有关这种转换处理的方式，展开为下面第4点讨论。

4. `XML::Simple`会观察嵌套元素（像之前看到的）是否含有特定属性，并由此做出相应处理。默认情况下，如果包含属性`name`、`key`或`id`，`XML::Simple`就会以该属性作为键构造哈希的哈希返回。所以上例中`<interface></interface>`要求转成数组时，它发现其内含`name`属性，便自动将其作为键，返回哈希。表面上看，它并非严格“按我说的去做”，不过最终我们得到的是更方便使用的数据结构。只要理解了它如何以及何时做此转换，使用上便不成问题。在我们的例子中，现在可以用这种方式表述：

```
# 后面两个箭头实际上可省略
$config->{hostname}->{interface}->{interface_name}
```

代码直观而清晰。

结合KeyAttr => {}, 我们还可以修正ForceArray的行为。与其依次遍历每个接口找出感兴趣的IP 地址, 不如直接改成通过IP地址访问接口信息, 这样就可以用此代码形式:

```
# 后面两个箭头实际上可省略
$config->{hostname}->{interface}->{ip_addr}
```

来访问这样的数据结构:

```
'interface' => HASH(0xa31c50)
  '192.168.0.101' => HASH(0xa31b18)
    'arec' => 'zeetha.example.edu'
    'name' => 'en0'
    'type' => 'Ethernet'
  '192.168.100.101' => HASH(0xa31b30)
    'arec' => 'zeetha.wireless.example.edu'
    'name' => 'en1'
    'type' => 'AirPort'
```

所以解析参数应当改成:

```
my $config = XMLin(
    'config.xml',
    NormaliseSpace => 2,
    ForceArray => ['interface'],          # 使用方括号列出元素名称
    KeyAttr    => { 'interface' => 'addr' }, # 使用大括号列出元素名称及对应属性
);
```

可以看到, KeyAttr选项将interface子元素按IP地址为键构造哈希返回时, 二级哈希中并不包含IP地址。如果要在该哈希中保留addr字段以方便使用, 可以在KeyAttr的属性名前加+表示。最后需要说明的一点是, 留着ForceArray的目的是要让单个<interface></interface>元素也按数组返回, 继而由KeyAttr一并转换键名。

从上面我们对XML::Simple的初步探索中可以看出, 就算是最简单的默认用法, 其解析处理也还是会遇到各种复杂情况的挑战。XML::Simple甚至提供了一种“严格”模式(就如同usestrict;)引导用户正确使用, 但这也不能完全保证不出错, 我们仍然需要小心处理各种潜在的问题。这种情况在来回读写XML文件时(即读取配置, 修改数据, 再写回文件)尤其明显和突出。

要修改XML的数据很容易, 只需像往常一样添加、删除、修改Perl数据结构中的某个元素即可, 但所有改动都在内存中, 接下来该如何把这些数据写到文件呢?

继续之前的代码范例，把下面这行添加到代码尾部：

```
print XMLout($config);
```

得到类似这样的输出：

```
...
<host name="agatha" os="linux" type="server">
  <interface name="eth0" arec="agatha.example.edu"
  cname="mail.example.edu" type="Ethernet" />
  <service>SMTP</service>
  <service>POP3</service>
  <service>IMAP4</service>
</host>
...
```

得到的结果几乎和我们开始时完全不同，而且还丢了数据（接口的IP地址）！如果按照模块的严格模式的建议，在XMLin()函数中添加了KeyAttr选项后，数据是回来了，但已不再是原来子元素的形式，而全都成了属性：

```
...
<host name="agatha" os="linux" type="server">
  <interface addr="192.168.0.4" arec="agatha.example.edu"
  cname="mail.example.edu" name="eth0" type="Ethernet" />
  <service>SMTP</service>
  <service>POP3</service>
  <service>IMAP4</service>
</host>
...
```

很尴尬的处境，不是吗？如果坚持选用XML::Simple的话，倒是有些变通的选择：

- 修改我们原来的数据文件格式。当然，这么做有点极端。
- 不断微调XML::Simple说明解析文件时的选项直至满足需求。比如，XML::Simple说明文档中建议针对这种情况应该在读写文件时都启用KeyAttr => {}选项。不过鱼和熊掌不可兼得，读写是方便了，但没有结构直观的哈希，我们查询和处理数据的工作又变得麻烦了。
- 在写入数据前调整。我们可以按照喜欢的结构读入数据（正如之前所做的），按需要处理后，再转成XML::Simple“喜欢”的数据结构写入外部文件。技术上并不难，只要对复杂数据结构的操作了然于胸，总是可以实现目标的。这种转换操作经常会用到若干map()函数，以及一堆看似复杂的操作符号。

最佳方集要取决于哪种（或任何一个）办法更适合你的具体情况。三种办法我以前都用过，哪种有用就用哪种。不过坦白说，有时候没有一种办法可以恰如其分地解决问题，只好抛开XML::Simple的简捷，完全改用其他模块。接下来我们就来讨论这种情形。

注意：Grant McLean，XML::Simple模块的作者，建议大家：

- 使用XML::Simple的严格模式来避免常见错误。
- 如果用XML::Simple花了5~10分钟还没有得到解析结果，换用XML::LibXML模块应该会快很多。

关于该模块他还写了一篇有意思的文章，可以到http://www.perlmonks.org/index.pl?node_id=490846阅读。

用XML::LibXML解析

XML::LibXML的优缺点

优点是：

- 解析器极为迅捷。
- 与标准规范兼容（目前仅支持XPath 1.0；编写本书时尚未有计划支持XPath 2.0）。
- 同时支持XPath和DOM，定位和操作数据非常容易。
- 它是Perl官方默认推荐使用的XML解析工具（根据<http://perl-xml.sourceforge.net/faq/>）。
- 模块维护者尽取尽责，积极回应提问及错误报告。
- 额外的好处是它还能解析并（大量地）操作HTML文件就好像它是XML。

缺点是：

- 系统必须预先安装可用且兼容的libxml2库，该模块底层需要调用该库实现各项操作。
- 模块的文档大部分讲述所支持的特性和规范，而只有一小部分介绍如何使用。如果本来就对XML、DOM、XPath等概念或其他相关标准有所熟知，那么理解如何使用并非难事。但对缺乏基础知识的新手来讲，就很难从文档中快速起步使用。这是目前为止该模块最大的缺憾了。
- 默认会把所有数据一次读入内存。

那么何时该用此模块呢？XML::LibXML适用于绝大多数处理常规大小XML文档的应用场景。它运作如飞，按标准行事，用起来也很方便，只需要理解XPath或DOM的基本概念，便能运用自如。

由Petr Pajas维护的XML::LibXML模块历来为人们所推荐，它有许多强大的功能和选项。我在这里只是蜻蜓点水，介绍一些最常见的用法。更多相关的故事可以到本章末列出的资源清单查阅。

和XML::Simple一样，XML::LibXML默认会把整个XML文档一次全部读入内存，然后数据的内存中表示。但它又和XML::Simple不同，XML::LibXML所操作的并非原生Perl数据结构，而是像一棵树一样展开分叉的节点，模块提供了各种方法对这些节点进行操作。如果无法马上理解XML数据按树结构表示的概念，请在这里做个标记，暂停阅读本章后续内容，先阅读附录B，然后再回到这里继续。这很重要，因为接下来的段落就需要对该附录介绍的背景知识有所了解，否则无法理解。

XML::LibXML支持两种常见的定位XML数据在树中所处位置的方法：W3C文档对象模型（Document Object Model，DOM）以及XPath。^[注10]虽然我在自己的编程实践中倾向于用XPath而非DOM，因为XPath更精准更雅致，不过我还是会结合两种不同实现的例子分别作介绍。既然XML::LibXML允许你随时选用两种方式中的一种操作XML数据，那么哪种方式更自然更直观便可选用哪种。

我们先从DOM方式开始吧。为了便于比较DOM和XPath的差异，我们还是选用XML::Simple里用过的那份XML文档。出于环保考虑，这里就不再重复列出了。

不管是DOM还是XPath方式，XML::LibXML程序的开始都是一样的（无非是加载模块、创建解析对象实例、解析XML数据）：

```
use XML::LibXML;

my $prsr = XML::LibXML->new();

my $doc = $prsr->parse_file('config.xml');
```

要遍历整棵DOM树，得先从XML的根节点元素开始：

```
my $root = $doc->documentElement();
```

从该元素开始，我们有两种遍历方式：一种是显式逐层遍历；一种是让模块自动搜索。逐层遍历很简单，在当前节点上查询获取子节点，然后不断迭代：

```
my @children = $root->childNodes;

foreach my $node (@children){
    print $node->nodeName(). "\n";
}
```

注10：确切地说，是DOM Level 2 Core和XPath 1.0（至少写此书时如此）。

如果运行上面给出的XML::LibXML代码，会看到下面这样奇怪的输出：

```
#text
description
#text
host
#text
host
#text
host
#text
host
#text
host
#text
host
#text
host
#text
```

description和host这些行还好理解（因为我们的文档中有<description>?</description>和<host></host>这些元素），可那些#text节点是什么意思呢？难道默认名字是#text？如果在程序运行过程中仔细查看其中某个#text节点的内容，你会看到：

```
x $node->data
0 '
  '
```

或者，换一种更容易看清的方式：

```
x split(//,$node->data);
0 '
  '
1 ' '
2 ' '
3 ' '
4 ' '
  '
```

该节点由一个回车符和四个空格符组成。XML::Simple默认会去掉空白字符；而XML::LibXML则尽可能保留任何遇见的空白字符，因为从字面上来讲，很难说空白字符本身是否具有特定意义，所以会在树中保留为文本节点。确实觉得“空”文本节点让你分心，而且用不到的话，完全可以让解析器丢弃所有这类节点。只需在解析工作开始前设定解析选项：

```
$prsr->keep_blanks(0);
my $doc = $prsr->parse_file('config.xml');
```

于是我们得到的输出正如预期（所有<network></network>——也就是文档根节点——的子节点名字）：

```
description
```

```
host
host
host
host
host
host
```

为叙述更加简洁，后续章节均默认使用`keep_blanks(0)`选项。

实际遍历树的方法非常简单：依次遍历当前所在节点的所有子节点，如果任何一个子节点还有子节点（可以用`hasChildNodes()`检测），就进入继续遍历。人们常用递归程序实现树的遍历，基本思想可以参阅第2章里讨论树遍历的相关代码。要手动定位文档树中比较深层的节点，就必须逐层深入下去：

```
my $root = $doc->documentElement();
my @children = $root->childNodes();

my $current = $children[2]; # 第二个 <host></host> 元素
@children = $current->childNodes();
$current = $children[1]; # 第一个 <service></service> 元素
print $current->textContent(); # 'HTTP'

# 或者，用一系列变量引用下标串起来写（很恶心吧）：
print STDOUT (($root->childNodes())[2]->childNodes())[1]->textContent();
```

除了往下遍历外，我们还可以用 `nextSibling()` 进入侧旁的兄弟节点：

```
my $root = $doc->documentElement();
my @children = $root->childNodes();

my $current = $children[2]; # 第二个 <host></host> 元素
$current = $current->nextSibling(); # 移到第三个 <host></host> 元素
```

这些手动遍历树的代码看起来着实让人痛苦，既麻烦又晦涩，不过另外一种办法：让XML::LibXML帮我们直接定位到特定元素。如果知道当前元素的哪些子元素是我们感兴趣的，可以用`getChildrenByTagName()`返回所有这些元素节点，只需在参数中指定要返回的子元素名称。比如，想要获取某台主机的网络接口定义：

```
my $root = $doc->documentElement();
my @children = $root->childNodes();

my $current = $children[5]; # krops 机器上的<host></host>元素
my @interface_nodes = $current->getChildrenByTagName('interface');
```

这个类似`grep()`的函数帮我们节省了不少力气，不用再那么麻烦地层层深入寻找特定元素。如果文档树非常庞大，这种优势就更明显。和`getChildrenByTagName()`相关的还有`getElementsByTagName()`方法。`getElementsByTagName()`不仅搜索当前元素的子节点，还会遍历所有子节点的子节点。比如，要获取所有主机的网络接口定义，可以这么写：

```
my $root = $doc->documentElement();
my @interface_nodes = $root->getElementsByTagName('interface');
```

一旦找到需要的节点后，便可以取得所有文本子节点内容，我们把这些内容当作当前元素节点的内容（所以每个节点的内容实际上是由子节点的内容合并而来）：

```
foreach my $node ( @interface_nodes ) {
    $node->textContent(); # 返回所有文本子节点的内容
}
```

如果该节点还有属性，也可以方便地列出来：

```
foreach my $attribute ($node->attributes()){
    print $attribute->nodeName . ":" . $attribute->getValue() . "\n";
}
# 或者仅选取特定属性：
print $node->getAttribute('name') if $node->hasAttribute('name');
```

注意：元素的属性保存在与当前节点相关联的属性节点中，所以我们可以用 `nodeName()` 方法返回属性名称。属性节点并非子节点，这和保存当前元素节点“内容”^[注11]的文本节点不一样。也就是说，如果调用 `childNodes()` 方法是不会返回属性节点的，但它会返回文本节点。不管是DOM还是XPath，这种情况都一样。

要修改元素节点的“内容”，就得修改其文本节点中的数据：

```
# 如果我们知道当前节点只有一个子节点，并且该节点就是文本节点，
# 直接对此节点操作即可。要判断是否为文本节点，按下面第一行代码测试。
#
# 如果不清楚子节点的情况，可以遍历 childNodes() 返回的节点，
# 测试 nodeName() 及 textContent()，然后再做类似修改。

my $textnode = $node->firstChild
if ($node->firstChild->nodeName == XML_TEXT_NODE);

$textnode->setData('new information');
```

有各种修改数据的方法可用，如 `insertData()`、`appendData()` 和 `replaceData()` 等，可按不同方式更新数据。而属性的修改则可直接通过类似 `getAttribute()` 的调用 `setAttribute()` 实现。

除了操作节点的数据外，有时还需要对节点本身或其代表的分支进行操作。比如新增或删除元素（及其子元素），就免不了要操作现有的XML文档上的节点。让我们从第二种

注11：之所以这里使用引号，是因为元素节点并非真正意义上拥有数据内容。只有它的文本子节点才保存着数据。不过当你看到 `<moo>baa-la-la</moo>` 这样的形式，总是会自然地认为其中的“baa-la-la”就是元素节点 `<moo></moo>` 的内容。

操作，即删除操作开始讲起，先讲简单容易的。删除一个节点（或者连带删除它下面所有子节点分支），需要先定位到该节点的父节点，然后让父节点移除该节点：

```
my $parent = $node->parentNode;
$parent->removeChild($node);
```

也可以两步并作一步，串起来写。执行完下面一行代码后，`$node`会被重新分配到一个父节点（`XML::LibXML::DocumentFragment`类型的节点）下面。

```
$node->parentNode->removeChild($node)
```

而新增元素节点则要稍稍麻烦些，因为在加入之前得先构造好该节点的所有信息。也就是说，我们必须先创建一个新的节点，设置属性，创建文本节点以及任何相关的子节点并赋值，一切准备妥当后才能挂载到原来的XML文档树中。

比如要添加一个新的`<meta></meta>`元素，并在其中放入若干描述该网络元信息的其他元素。比如`<type></type>`元素，用于区分该网络属于开发环境还是生产环境。这段XML看起来像这样（为提高可读性另外补充了空白字符）：

```
<meta>
  <type name="production">This is a production network</type>
</meta>
```

为了创建如上内容，我们的代码得这样写：

```
# 创建一个崭新的 XML 元素，设置属性和文本节点内容
my $type = $doc->createElement('type');
$type->setAttribute( 'name', 'production' );
$type->appendTextNode('This is a production network');

my $meta = $doc->createElement('meta');

# 将 <type></type> 作为子元素，或者说子节点，挂载到<meta></meta>下面
$meta->appendChild($type);
```

如果你是个锱铢必较的人（当然这是网络系统管理员的优良品质），一定会看不惯`<description></description>`元素独立于`<meta></meta>`外，网络描述也该是元信息的一种啊。好吧，让我们动手修改，把它移到meta标签内：

```
my $root = $doc->documentElement();

# 先从根元素开始定位<description></description>元素，
# 然后将其追加为<meta></meta>最后的子节点
$meta->appendChild($root->getChildrenByTagName('description'));
```

现在我们准备好了所需的XML文档片段，是时候挂到原来的文档树中去了。偷懒的办法

是直接将它追加为根节点的最后一个子节点（`$root->appendChild($meta)`），不过最好还是安置在文档开始的部分，好给读者解释文档的后续内容：

```
my $root = $doc->documentElement();

# 将 $meta 插入到根节点 $root 的第一个子节点之前
$root->insertBefore($meta,$root->firstChild);
```

要是打算插入一大堆这样的节点，还用这种原始手动方式一定会累死人。不必担心，XML::LibXML还提供了漂亮便捷的`parse_balanced_chunk()`函数，给它XML数据，它就会帮你返回相应的文档片段，再挂载到原来的文档树就方便多了。还是用之前的`<meta></meta>`范例演示该函数的用法：

```
my $root = $doc->documentElement();

my $xmloitinsert = <<'EOXML';
<meta>
  <type name="production">This is a production network</type>
</meta>
EOXML

my $meta = $prsr->parse_balanced_chunk($xmloitinsert);

$root->insertBefore($meta,$root->firstChild);
```

一旦内存中有了完整的文档树，不管之前如何修改节点或树分支，最后写入文件易如反掌：

```
open my $OUTPUT_FILE, '>', $filename or
  die "Can't open $filename for writing: $!\n";
print $OUTPUT_FILE $doc->toString;
close $OUTPUT_FILE;
```

以上就是最基本的操作DOM模型的方法。至此你应该对XML::LibXML有了大体了解，而对XML文档的理解也会更加深入透彻。接下来不妨看看XML::LibXML::{Element, Node, Attr及Text}文档中的说明，还有无数有趣的方法可以使用，我们无法一一在此介绍。

如果之前按照我的建议阅读了有关XPath附注的话，你一定好奇该如何把这些概念付诸实践。现在就来看看。

之前讲过，不管是用DOM还是XPath，XML::LibXML程序的开始部分都是相同的：

```
use XML::LibXML;

my $prsr = XML::LibXML->new();
$prsr->keep_blanks(0);
my $doc = $prsr->parse_file('config.xml');
```

第一个不同之处是搜索和定位节点树位置的方法。不用像之前那样手动逐层遍历到相应节点，直接在`findnodes()`方法里给出节点所在路径就可以了。为便于比较，我们用XPath重新做一遍之前用DOM完成的相同工作。首先是获取根节点的所有子节点。等效的XPath实现如下：

```
my @children = $doc->findnodes('/network/*');

foreach my $node (@children){
    print "$node->nodeName()\n";
}
```

我知道很简单，没什么可激动的。不过接下来的任务是要获取XML文件中第二台主机的第一项服务的具体内容，这在之前DOM的例子中我们可没少忙活，如果改用XPath方式的话，一般也就一两行代码直接搞定：

```
# 如果是标量上下文，findnodes()方法会返回一个NodeList对象，
# 但我们这里只需要列表中表示文本的节点，所以在$node两边使用圆括号，
# 强制 findnodes() 方法按照列表上下文返回数据
my ($node) = $doc->findnodes('/network/host[2]/service[1]/text()');
print $node->data . "\n";

# 或者，也可以按这种手法打印所有文本节点的数据：
foreach my $node ($doc->findnodes('/network/host[2]/service[1]/text()')){
    print $node->data . "\n";
}
```

只是一次`findnodes()`方法调用，我们便取得了所需节点对应的文本节点。`findnodes()`支持XPath 1.0的定位路径的所有功能。这种强大的定位功能还包括后代操作符（descendant operator，记为`//`），它基本上可以取代DOM里繁琐的`getChildrenByTagName()`以及`getElementByTagName()`调用，只需一次搜索便可取得所有感兴趣的节点——无论它在什么层次上（这还要感谢XPath的谓词）：

```
# 找出所有提供不止一项服务的主机
my @multiservers = $doc->findnodes('//host[count(service) > 1]');

# 或者直接取出这些主机的名字属性节点，然后打印主机名
foreach my $anode ($doc->findnodes('//host[count(service) > 1]/@name')){
    print $anode->value . "\n";
}
```

这里用XPath后代操作符找出文档中任意位置出现的主机元素节点，然后在谓词中用XPath的`count()`函数进行过滤。

现在我们来看一个简单的XPath范例，它充分展示了使用XML::LibXML编程的灵活和方便：

```
@nodes = $doc->findnodes('/network/host[@type = "server"]//addr');
```

这个XPath表达式将找出所有服务器并返回它们的<addr></addr>元素节点。实际上我们并不关心返回的节点本身，我们要的是存储在它们的文本子节点中的数据（也就是具体的地址）。在此基础上，我们可以有三种变通的做法，具体要看你喜欢哪种编程风格，或者程序上下文中使用数据的方式：

1. 修改上面的XPath表达式（添加text()部分）：

```
@nodes = $doc->findnodes('/network/host[@type = "server"]//addr/text()');
```

这会返回所有保存地址信息的文本子节点。我们在前面已经见过如何循环迭代列表中的每个节点，并用data()方法调用取出数据，所以这里就不再重复演示foreach()循环代码了。

2. 在返回的节点上依次作进一步的XPath计算：

```
foreach my $node (@nodes){  
    print $node->find('normalize-space(./text())') . "\n";  
}
```

请注意，这里改用find()而不是之前的findnodes()，是因为我们要直接提取符合条件节点的字符串内容（而不是返回一个节点对象或一组节点对象）。这里的表达式就好比是说：“从当前节点开始，找出所有相关的文本节点，并规范化它们的值，去掉出现在首尾的空白字符，然后返回。”当然，我们可以省去normalize-space()函数调用，保留原始的空格字符，再用Perl做后续处理。不过这里想要展示的是，把任务分解成两个XPath调用，使得代码中的定位路径更加清晰。

3. 此时改用DOM方式：

```
foreach my $node (@nodes) {  
    print $node->textContent() . "\n";  
}
```

最后一种办式看似貌不惊人，但实际上这种思想在我们的XPath讨论中尤为重要。XPath擅长于定位和查询文档中满足一定条件的节点，但它并不善于修改或处理文档内容。一旦确定了感兴趣的节点或节点树之后，XPath便可以功成身退，改由DOM披挂上阵。之前我们所见的对节点或节点树的DOM操作，此刻都可以直接拿来使用。比如，要修改节点的文本内容，直接用setData()；如果要删除节点，从它的父节点调用removeChild()；甚至用to_string()^[译注3]直接输出XML字串也是一样的。

译注3：按最新的1.70版，应该是toString()。

XPath和XHTML

这里是XML::LibXML模块当前维护者Petra Pajas推荐我和大家分享的小贴士：

开始用XPath解析XHTML文档的新手（当然是用XML::LibXML）经常会倍感受挫，用最简单的XPath定位路径如/html/body都得不到匹配的节点。类似这样的问题不断出现在perl-XML邮件列表中，也难怪，看起来显然可以得到结果的操作却无法正常工作，谁碰到都会纳闷。

这里解释一下原因：XHTML文档在开始的地方通常会有一个默认的关于名称空间的定义（<html xmlns="http://www.w3.org/1999/xhtml">）。更完整的说明参阅第227页上的补充内容“XML名称空间”。假如我们用Perl术语来解释，可以理解为<html></html>和<body></body>元素都位于另外一个不同的包里面，而一般XPath解析器开始搜索的地方并不在这个包里。所以，为了切换到新的名称空间，我们需要给XPath分配一个名称空间前缀以便映射过去，然后在搜索的时候给出这个前缀即可。比如，用/x:html/x:body这样的写法就能正常工作了。

在XML::LibXML里创建映射之前，得先准备好XPathContext对象（在此基础上进行XPath相关的操作），然后用该对象注册一个新的XHTML名称空间前缀。下面的代码演示了如何操作，所搜索的是XHTML文档中表示文字段落的部分，并依次打印其内容：

```
use XML::LibXML;
use XML::LibXML::XPathContext;

my $doc = XML::LibXML->new->parse_file('index.xhtml');
my $xpath = XML::LibXML::XPathContext->new($doc);

$xpath->registerNs( x => 'http://www.w3.org/1999/xhtml' );

for my $paragraph ($xpath->findnodes('//x:p')) {
    print $paragraph->textContent, "\n";
}
```

希望这个小贴士对你有所帮助。

为透彻理解并巩固之前学到的知识，现在让我们来看一个现实世界中更为复杂的混合使用XPath/DOM的例子。我们将为上面这份XML文件描述的网络生成一份DNS域名配置文件。这里需要关注的是XML结构本身，所以在生成正确的域名配置文件头时，我们直接借用了第5章中的GenerateHeader子例程：

```
use XML::LibXML;
use Readonly;
```

```

Readonly my $domain => '.example.edu';

# 引自第5章中的程序
print GenerateHeader();

my $prsr = XML::LibXML->new();
$prsr->keep_blanks(0);
my $doc = $prsr->parse_file('config.xml');

# 找出所有通过以太网方式连接的机器的网络接口节点
foreach
{
    my $interface ( $doc->findnodes('//host/interface[@type = "Ethernet"]') )

    # 通过 DOM 方法获取每台机器的信息，
    # 继而合为一个注解并打印
    my $p = $interface->parentNode;
    print "\n; "
        . $p->getAttribute('name')
        . ' is a '
        . $p->getAttribute('type')
        . ' running '
        . $p->getAttribute('os') . "\n";

    # 打印每台主机的 A 记录
    #
    # 当然，我们可以用Perl正则表达式去掉域名部分或是首尾的空白字符（而且这么做可能更直
    白清楚），
    # 不过这里的例子是要说明 XPath 函数的使用
    my $arrname = $interface->find(
        " substring-before( normalize-space( arec / text() ), '$domain' ) ");

    print "$arrname \tIN A \t \t "
        . $interface->find('normalize-space(addr/text())') . " \n ";

    # 找出所有的 CNAME RR 记录并打印
    #
    # 在同一个循环中混用DOM和XPath方法时需要注意：
    # XPath 调用一般都有较多计算，所以应该尽量避免在嵌套循环内部反复使用XPath函数，
    # 特别是在生产代码中更应小心
    foreach my $cnamenode ( $interface->getChildrenByTagName('cname') ) {
        print $cnamenode->find(
            " substring-before(normalize-space(./text()), '$domain' )"
            . "\tIN CNAME\t$arrname\n";
    }

    # 我们还可以做得更多，比如输出SRV 记录等等。
}

```

到此为止，暂且把基于树结构的XML解析放到一边吧，继续我们的XML之旅。

通过XML::SAX的SAX2的优缺点

优点是：

- 边接收数据边解析（不必等到所有文档数据都传送到位后再开始解析）。
- SAX2规范有多种语言的支持（SAX最初是从Java世界发展而来的，不过很快几乎被所有主流脚本语言所采用。这就是说，用Perl写的SAX2代码，至少在概念层上，很容易被其他写Java、Python、Ruby的同事理解。）
- 借助XML::SAX我们可以只用一套基础代码，但用不同的后端解析器处理。
- XML::SAX完全使用面向对象方式工作。
- SAX2还有一些非常酷的高级特性，比如管道传送（pipelining，多个XML过滤例程串联）以及从非XML源中取用数据或者从XML导出数据等等。

缺点是：

- 瞬息万变，不可轻心。解析器只会在特定时刻报告信息，如果事先未加留意保存或者想到改用其他数据结构保存接收到的信息，那么为时已晚。
- XML::SAX完全面向对象。所以如果你的编程经历缺乏这方面的经验，那么学习曲线或许会比较陡峭。
- 有些操作必须依赖人手工完成，所以有时还要额外写代码。例如收集文本类型数据以及查找特定元素等等都是如此。想要保存任何解析后得到的数据结构（树），就只能自己动手写代码实现。

那么何时该用XML::SAX呢？对于数据量庞大的XML文件，凡是不适合一次全部读入内存的，都应该使用这个模块。而且XML::SAX更适合用在事件驱动的编程模型中，只要有需要，立即边解析边给出结果。如果你原来使用了XML::Parser，那么无疑这是接下来的目标。

到目前为止，我们所见到的处理XML文件的方式都需要一次读入所有数据到内存，然后再解析。即便现在内存的价格日趋廉价，也不能成为铺张的理由。况且要是手头的XML数据集极其庞大，全部放入内存未必能正常工作。此外还有时耗和效率的问题，有些简单的事情必须要等到所有数据加载完成后才从头到尾做一遍完整解析，最后才有所响应的话，这对用户而言就太糟糕了。有时候数据尚未传送到位（比如来自网络），就只能干等，无法开始解析工作。而且多数时候，我们要的只不过是文档做一些简单的修改（比如把所有<service>?</service>元素改名为<protocol></protocol>，或者类似这

样的修改），全部读入并解析的话，就会多出许多不必要的工作来。基于树模型的解析方式，就必定要逐个读入每个元素（就算它和我们关心的`<service></service>`毫不相干也必须一一读取），每个元素都平等对待，耗费一样的计算资源处理，这显然是在做无用功。

接下来我们要讨论的正是解决以上缺点的另一种XML处理标准模型：SAX。SAX字面上表示XML的简单应用编程接口（Simple API for XML），现在已经发展到第二个主要版本（SAX2）。它所提供的处理模型将XML文档数据当作一系列事件流来处理。可能这么讲你难以理解，让我们先回顾下Perl/XML历史，然后就会明白。

很久很久以前，XML工作组（XML Working Group）的技术带头人James Clark用C语言编写了一个非常时髦漂亮的XML解析库，称之为`expat`。`expat`库是当时的经典，随着XML的流行，人们开始把它广泛用于各式项目来完成XML文档的解析任务（到写本书时，Apache HTTP Server和Mozilla的Firefox浏览器等重要项目都还在用）。Larry Wall自己还写过一个从Perl里面调用`expat`的模块`XML::Parser`，后由Clark Cooper修改并维护了好多年。目前是Matt Sergeant在维护。

`XML::Parser`提供了几种操作XML数据的方式，我们来快速看一下其`stream`风格（或者说解析模式），明白由此衍生出的概念后再回到之前关于SAX2的讨论，就会好理解多了。

首先来了解技术背景。`XML::Parser`是一个基于事件的模块，这就好比是雇用股票经纪人买卖股票一样，在交易开始之前告诉她一系列行动指引，让她根据股市变化采取相应行动（比如每股价格低至3.25美元时就抛出一千股，或者在每个交易日开始的时候就买进等等）。在事件基于驱动的编程中，触发行动的叫做事件（event），因这个事件而采取的行动则称为事件处理器（event handler）。其实所谓的不过是在特定事件发生时才调用的特殊例程，所以也有人称之为回调例程（callback routine），就好像原来的程序在满足特定条件时才“回来调用该例程”一样。对于`XML::Parser`模块而言，接踵而至的事件无非就是类似于“开始解析数据流”、“发现一个起始标签”、“找到一个XML注释”等，然后对应的事件处理器将做一些类似“打印当前元素内容”的事情。

在开始解析数据之前，得先构造一个`XML::Parser`对象。而在创建该对象时就需要指定一种解析模式，或者说风格。`XML::Parser`支持多种风格，每种都略有不同，不同的风格决定了默认情况下所调用的事件处理器以及返回数据时的构造方式。

某些风格还需要我们手动指定所关心的事件及其处理方式，没有显式声明的就不会采取任何行动。这些事件和事件处理器保存在一个简单的哈希表中作为参数传入，其键为

要处理的事件的名称，其值为事件处理器例程的引用。创建解析器对象时，我们使用 `Handlers` 参数来传入哈希（比如，`Handlers => {Start => \&start_handler}`）。

我们接下来要用的自然是 `stream` 风格，它不需要手动指定要处理的事件：它只是调用一组预定义的事件处理器，如果在程序的名称空间中发现了例程的话。我们要用到的 `stream` 事件处理器很简单：`StartTag`、`EndTag` 以及 `Text`。除 `Text` 外的两个事件名称都可以自我说明，而根据 `XML::Parser` 的说明文档，`Text` 表示“在 `$_` 变量中还留有不含标记的文本的任何起始或结束标签之前调用”这种情况，所以在需要知道某个元素的内容时，就得用 `Text` 事件。先来看代码吧，然后再来解释其中有意思的部分：

```
use strict;
use XML::Parser;
use YAML; # 仅用于显示结果，并非解析需要

my $parser = new XML::Parser(
    ErrorContext => 3,
    Style        => 'Stream',
    Pkg          => 'Config::Parse'
);

$parser->parsefile('config.xml');
print Dump( \%Config::Parse::hosts );

package Config::Parse;

our %hosts;
our $current_host;
our $current_interface;

sub StartTag {
    my $parser = shift;
    my $element = shift;
    my %attr    = %_; # 注意，不是 @_, 见 XML::Parser 文档

    if ( $element eq 'host' ) {
        $current_host      = $attr{name};
        $hosts{$current_host}{type} = $attr{type};
        $hosts{$current_host}{os}  = $attr{os};
    }

    if ( $element eq 'interface' ) {
        $current_interface = $attr{name};
        $hosts{$current_host}{interfaces}{$current_interface}{type}
            = $attr{type};
    }
}

sub Text {
    my $parser      = shift;
    my $text        = $_;
    my $current_element = $parser->current_element();
```

```

$text =~ s/^\s+|\s+$//g;

if ( $current_element eq 'arec' or $current_element eq 'addr' ) {
    $hosts{$current_host}{interfaces}{$current_interface}
        {$current_element} = $text;
}

if ( $current_element eq 'cname' ) {
    push(
        @{ $hosts{$current_host}{interfaces}{$current_interface}{cnames}
            },
        $text
    );
}

if ( $current_element eq 'service' ) {
    push( @{ $hosts{$current_host}{services} }, $text );
}

}

sub StartDocument { }
sub EndTag        { }
sub PI           { }
sub EndDocument  { }

```

该程序的大部分工作由StartTag()和Text()这两个子例程完成。如果碰到 <host> 起始标签，就在哈希添加一条记录，在标签属性中取出主机名作为新记录的键，然后取出其他标签属性作为子哈希保存。程序开始的时候我们就设置了一个全局变量来保存当前<host></host>元素对应的主机名称，以便在接下来处理嵌套的其他标签时可以使用。比如遇到<interface></interface> 元素的起始标签时，便可以把网络接口相关的信息保存到原来主机下面的哈希结构中，然后同样地将当前网络接口保存到全局变量中，以便我们在更深层次标签处理时能感知当前所在的网络接口。这种利用全局变量来维护当前进入的元素层次的做法，在使用XML::Parser的程序中非常普遍，但考虑到可维护性的话，这绝不是一种漂亮的解决方案（因为“不加限制的全局变量没有保障”）。XML::SAX模块的教程指出，最好使用闭包维护XML::Parser中用到的状态变量，不过真要这么做，代码可就复杂多了，这里只是概念演示，不必过于纠结。

我们关心的元素内容则由Text()子例程负责处理。像<arec></arec>和<addr></addr>这样只在每个网络接口中出现一次的元素，可以直接保存为适当接口子哈希中的一对键值。所谓适当接口，就是之前StartTag()处理时所设置的全局变量表示的。而后续处理<cname></cname>和<service></service>标签的代码则要稍稍复杂些，因为同一个主机的网络接口下面可能有多项记录，所以保存时应该使用匿名数组，逐个推入后挂在原始哈希表下。

这段代码还有两个有意思的地方，一个是尾部的空子例程，另一个是`StartTag()`和`Text()`生成的数据结构的显示方式。之所以保留空子例程，是因为使用`stream`风格的`XML::Parser`在遇到未定义事件处理器的标签时会默认打印其内容。这里我们不希望它输出任何数据，所以定义了空子例程来回避。

所构造的数据结构最后采用YAML输出。下面节选其中的一部分：

```
agatha:
  interfaces:
    eth0:
      addr: 192.168.0.4
      arec: agatha.example.edu
      cnames:
        - mail.example.edu
      type: Ethernet
  os: linux
  services:
    - SMTP
    - POP3
    - IMAP4
  type: server
...

zeetha:
  interfaces:
    en0:
      addr: 192.168.0.101
      arec: zeetha.example.edu
      type: Ethernet
    en1:
      addr: 192.168.100.101
      arec: zeetha.wireless.example.edu
      type: AirPort
  os: osx
  type: client
```

本章稍后会关注有关YAML的介绍，这里就当作伏笔先知道一下好了。

现在差不多可以回到SAX2的讨论了。和`XML::Parser`的`stream`风格类似，SAX2也是一个基于事件的API，由用户提供处理特定事件的代码，在进行XML解析时按事件触发调用。`XML::Parser`和`XML::SAX`的主要差别在于后者是完全面向对象操作的。没有面向对象编程经验的人或许要犯难了，不必担心，我会尽量用最简单的语言介绍`XML::SAX`范例中面向对象编程的概念。要是你确实想系统学习Perl的面向对象编程，Damian Conway的《Object Oriented Perl: A Comprehensive Guide to Concepts and Programming Techniques》（Manning出版）绝对是不二的选择。另外需要注意的是，这里的介绍仍然只是浮光掠影，蜻蜓点水。深入且高级的SAX2概念和用法，请参考本章末的文献索引。

废话少说，先来看代码。我们要写的代码有两部分：一部分是XML::SAX解析器初始化代码；第二部分是事件处理器。最简单的解析器可由XML::SAX::ParserFactory返回实例对象：

```
use XML::SAX;
use YAML;          # 仅用于显示结果，并非解析需要

use HostHandler;    # 稍后我们来定义该模块

my $parser = XML::SAX::ParserFactory->parser( Handler => HostHandler->new );
```

表面来看，这段代码中有两点不是很好理解。首先是加载了HostHandler模块，这个是我们稍后就要构建的，我会在其中安置所有事件处理器的实现。我把它称作HostHandler是因为它将提供处理器对象，在解析器处理主机定义中的SAX2事件时使用。^[注12] 该模块的new()方法会返回封装了所有事件处理器的对象。如果暂时无法理解，先放一放。稍后看过具体代码范例后再回来看就会明白。

让我们回到先前的解析器初始化代码。第二个不太容易理解的是代码中所调用的竟然是名字超长的XML::SAX::ParserFactory模块。该模块的目的（这里我是故意不用面向对象术语）是要从合适的解析器提供的模块（parser-providing module）返回具体的解析器对象。解析器提供的模块包括XML::LibXML和XML::SAX::PurePerl，而纯Perl解析器则是包装在XML::SAX内部的。XML::SAX::ParserFactory提供了一般化的接口返回所需要的解析器对象，所以无论底层用哪种XML::SAX兼容的解析器提供的模块都没关系，应用层的代码始终是一致的。上面的例子中，我们让XML::SAX::ParserFactory随意选取其一，当然也可以特别指定（具体方法请参考该模块说明文档）。

一旦准备好解析器，接下来就是要读取并分析XML文档了：

```
open my $XML_DOC, '<', 'config.xml' or die "Could not open config.xml:$!";

# parse_file 方法所接受的是文件句柄，而非普通文件名
$parser->parse_file($XML_DOC);

close $XML_DOC;

print Dump( \%HostHandler::hosts );
```

现在来看看SAX内部实际要做的事情，也就是具体的事件处理代码。我们会示意性地给出部分代码并和之前的XML::Parser进行比较。在XML::Parser里，随着文档的读取，标签会触发事件，我们为这些事件编写了处理子例程。而在SAX里，事件的名字有

注12：原则上模块名字可以随便取，哪怕叫做 BobsYourUncle 也成，不过我还是建议用有意义的名字，否则会让其他阅读代码的人难以理解。

一点儿不同：StartTag()变成了start_element(), EndTag()变成了end_element(), 而Text()（大多数）也变成了characters()。

两组子例程有一个显著的差别：XML::Parser里用的子例程定义在哪个包中都可以，没有特别要求；而XML::SAX的子例程则必须是类方法。没听到过“类方法”这种面向对象讲法也没关系，不必担心，在XML::SAX里做起来其实很简单。所需做的，仅仅是在包开头添加两行代码，之后所有子例程就都变成类方法了（更确切点的说法是，现在开始覆盖了原来XML::SAX::Base提供的默认方法）：

```
package HostHandler;
use base 'XML::SAX::Base';
```

XML::SAX::Base 负责处理有关解析器对象的基础构建工作，包括我们在初始化代码中调用的new()方法等。

注意： 如果以前没写过这样的代码，现在就是个好机会，转换思维，习惯一下面向对象编程中的思考方式。无需华丽辞藻，只需简单地记住一点：写完这两行，就会有一个解析器对象，它会替我们负责封装数据和行为，而我们所要做的，仅仅是在类方法中给出具体的行动方案。

在解析器对象所拥有的代码中，有一部分子例程会在遇到某些特定事件时被调用，这些子例程属于对象方法。比如遇到起始标签，解析器对象就会调用它的start_element()子例程（或者说方法）。此外，对象中还有一些辅助子例程，可以作为内部使用的工具。我们还可以让对象保存有关于它自身的一些数据（如当前正在解析的主机名等等），这就免去了前一节用全局变量保存状态的粗陋做法。

基本上可以了，接下来的内容所需要的面向对象知识也就是这些了。

让我们来看看这些方法的具体定义吧。下面是解析器对象遇到某个元素的起始标签时所调用的方法：

```
# %hosts 用于收集所有解析后得到的数据
# （是的，我们也可以把它保存到对象内部）
my %hosts;

sub start_element {
    my ( $self, $element ) = @_;

    $self->_contents('');

    # 此处的 '{}something' 写法令人惊奇，这是 James Clark 的记法。
    # 稍后谈到 XML 名称空间时，我们会作解释
    if ( $element->{LocalName} eq 'host' ) {
        $self->{current_host} = $element->{Attributes}->{{}name}{Value};
        $hosts{ $self->{current_host} }{type}
            = $element->{Attributes}->{{}type}{Value};
    }
}
```

```

        $hosts{ $self->{current_host} }{os}
            = $element->{Attributes}{'{os}'}{Value};
    }

    if ( $element->{LocalName} eq 'interface' ) {
        $self->{current_interface} = $element->{Attributes}{'{name}'}{Value};
        $hosts{ $self->{current_host} }{interfaces}
            { $self->{current_interface} }{type}
            = $element->{Attributes}{'{type}'}{Value};
    }

    $self->{current_element} = $element->{LocalName};

    $self->SUPER::start_element($element);
}

```

显然，这个子例程改自XML::Parser范例中的那个等价子例程，所以只要看看修改之处就行了。首先是传入参数有变化。XML::SAX在调用事件处理器时，会把对象引用作为第一个参数，而把该方法所需的其他特定信息作为后续参数传入。于是，start_element()便可从第二个参数获知当前解析器遇见的是哪个元素，该参数实际是元素数据结构的哈希引用，所以展开后就像这样：

```

0 HASH(0xa30624)
  'LocalName' => 'host'
  'Name' => 'host'
  'NamespaceURI' => undef
  'Prefix' => ''
  'Attributes' => HASH(0xa30768)
    '{name}' => HASH(0xa3033c)
      'LocalName' => 'name'
      'Name' => 'name'
      'NamespaceURI' => ''
      'Prefix' => ''
      'Value' => 'agatha'
    '{os}' => HASH(0xa307f8)
      'LocalName' => 'os'
      'Name' => 'os'
      'NamespaceURI' => ''
      'Prefix' => ''
      'Value' => 'linux'
    '{type}' => HASH(0xa30678)
      'LocalName' => 'type'
      'Name' => 'type'
      'NamespaceURI' => ''
      'Prefix' => ''
      'Value' => 'server'

```

该哈希的各项字段意义在表6-1中说明。

表 6-1: 传入 start_element() 的哈希内容

哈希键	对应内容
LocalName	元素的名称, 不包含名称空间前缀 (关于名称空间前缀的概念, 请参见补充内容“XML 名称空间”)
Name	元素的完整名称, 包含名称空间前缀
Prefix	该元素的名称空间前缀 (如果有的话)
NamespaceURI	该元素的名称空间所对应的 URI (如果有的话)
Attributes	由哈希组成的哈希, 保存着该元素所有的属性信息

XML 名称空间

在此之前, 我都有意回避提及任何有关 XML 名称空间的概念。一般小型的 XML 文档 (比如配置文件) 都不会用到, 况且我也不想在接下来的章节中徒增繁复。不过既然 XML::SAX 为它的事件处理器提供了名称空间信息, 我觉得至少应该对此有个大概了解会更妥当。至于更加深入具体的细节, 可以查阅 W3C 官方介绍, 位于 <http://www.w3.org/TR/REC-xml-names/>。

XML 名称空间是确保文档中元素唯一且能相互区分的方法。如果有份文档使用 `<orange></orange>` 元素, 那么它和它的子元素所表示的可能是颜色, 也可能是水果。但在特殊情况下, 比如某间设计公司给种橘子的客户提供新的果汁包装盒创意时, `<orange></orange>` 元素便可能会产生歧义。有了名称空间, 你可以添加一个特殊属性 (`xmlns`), 用于区分语义范围:

```
<orange xmlns="http://colors.example.com/chart"> ... </orange>
```

现在文档中所有元素都与此名称空间所表示的 URI^[注13] 相关联, 这样谁都清楚, 其中的 `orange` 是指颜色。^[注14]

注13: 此处的 URI 只是一种简便的表示唯一字符串的方式, 这样名称空间就不会有重复。所以, 该 URI 地址不一定是真实可访问的, 解析器永远都不会建立连接来请求此页面资源。不过把相关的文档地址作为 URI 倒是很酷的做法 (比如<http://www.w3.org/1999/XSL/Transform>), 当然这也不是必须的。

注14: 这么说可能好懂些, 把 XML 的名称空间理解为 Perl 的 `package` 语句就好了。 `package foo` 之后的所有代码 (直到另一条 `package` 语句之前) 都会归入 `foo` 名称空间。这样便可以拥有两个同名标量 `$orange`, 一个在你当前的程序中^{译注4}, 一个在所定义的包的名称空间中。

译注4: 当前程序的名称空间为 `main`。

对于同一个元素，我们还能定义多个名称空间，写法上稍微复杂些，每个都要跟上相应标识字符串（或称前缀）：

```
<juicebox xmlns:color="http://colors.example.com/chart"
          xmlns:fruit="http://fruits.example.com/fruitlst">
  <color:orange>#ffa500</color:orange>
  <fruit:orange>Citrus sinensis</fruit:orange>
</juicebox>
```

上面定义了两个不同的名称空间，分别用前缀color和fruit表示。之后，我们便可以在两个<orange></orange>元素中分别按照namespace: orange这样的语法加注名称空间，如上面代码所示，这样就不会相互混淆了。

还有一点要注意：James Clark，这位在XML领域有诸多杰出贡献的作者（也就是之前所提到的expat解析器的作者），为显示名称空间独创了一种非正式的语法，大家都称其为“James Clark记法”。它基本上是用<{namespace}element_name>这样的形式，所以前面第一个<orange></orange>元素可以写成：

```
<{http://colors.example.com/chart}orange> ... </orange>
```

这种写法不被任何XML解析器所接受，不过XML::SAX会在元素属性的哈希键中借鉴这种记法。

如果某个元素包含一些属性（见之前的范例），那么这些属性会以哈希的哈希表示。此哈希中的每个键为各个属性的名称，用James Clark记法表示（参见之前的补充内容），键的对应内容为表示所有属性信息的哈希，见表6-2。

表6-2：用于存储属性信息的哈希的内容

哈希的键	对应内容
LocalName	属性的名称，不包含名称空间前缀
Name	属性的完整名称，包含名称空间前缀（如果有的话）
Prefix	属性的名称空间前缀（如果有的话）
NamespaceURI	该属性的名称空间所对应的URI（如果有前缀的话）
Value	属性的值

我们的配置文件一般都不用名称空间，所以这里的数据结构总是以空的前缀（{}）表示各项属性名称。这会让哈希键看起来有点奇怪。

明白了某个元素的信息是如何传入start_element()方法之后，再回过头来看前面展示的程序代码，应该会有更透彻的理解了吧。除了_content()和SUPER::start_element()

方法之外（稍后会来解释这两个），所有的代码本质上所做的无非就是从`$element`的数据结构中复制一些东西出来，然后保存到我们的`%hosts`哈希中去；或者从`$element`中提取关键信息（比如当前元素名称）存入解析器对象^[注15]中以备用。

以上就是解析器碰到新标签时所做的事情。接下来我们看看遇到元素的文本内容（而不是另一个子元素）时是如何处理的：

```
sub characters {
    my ( $self, $data ) = @_;

    $self->_contents( $self->_contents() . $data->{Data} );

    $self->SUPER::characters($data);
}
```

你一定注意到这比之前`XML::Parser`范例里的`Text()`子例程短了不少，它只是用一个独立的 `_contents()`方法^[注16]逐个串接传入的数据罢了（暂时跳过神秘的第二行`SUPER::`，很快我就会解释），该方法定义如下：

```
# 如果有任何文本传入，则存入当前对象的内部数据结构中；
# 但不管怎样，最后都会返回当前所存内容
sub _contents {
    my ( $self, $text ) = @_;

    $self->{'_contents'} = $text if defined $text;

    return $self->{'_contents'};
}
```

`characters()`方法比`Text()`子例程小得多，这是因为两个模块的工作方式不同，这种差异直接导致了最终行为上的变化。使用`Text()`时，模块作者可以保证它会一次得到元素内（按文档中的原话来说）“积累至今所有不含标签的文本”。而对`characters()`来说，情况并非如此。`XML::SAX`教程中提到：“SAX 解析器天生无法保证 XML 文档中对某个标签内一段文本调用多少次`characters`方法——或许只需一次，或许文本内的每个字符都要一次。”所以，像`XML::Parser`代码中一次取出元素内容的做法在这里行不通。只能等到最后触发`end_element()`事件处理器时，才能确定已经收全所有文本内容，然后才能处理。在`end_element()`事件处理器里头，第一件要做的事情就是取出元素内容，移除首尾空白字符，存好备用：

注15：面向对象的纯粹主义者看到这里，或许会用钢头靴踹我，在对象中保存和取用数据怎么可以不用“getter”和“setter”方法呢？好吧，我只是希望尽可能简化代码，集中注意力在`XML::SAX`模块上，不过你的观点我接受，现在没问题了吧。

注16：我想现在那些面向对象圣徒该满意了吧……

```

sub end_element {
    my ( $self, $element ) = @_;

    my $text = $self->_contents();

    $text =~ s/^\s+|\s+$//g;    # 移除首尾空白字符

    if ( $self->{current_element} eq 'arec'
        or $self->{current_element} eq 'addr' )
    {
        $hosts{ $self->{current_host} }{interfaces}
            { $self->{current_interface} }{ $self->{current_element} }
            = $text;
    }

    if ( $self->{current_element} eq 'cname' ) {
        push(
            @{ $hosts{ $self->{current_host} }{interfaces}
                { $self->{current_interface} }{cnames}
            },
            $text
        );
    }

    if ( $self->{current_element} eq 'service' ) {
        push( @{ $hosts{ $self->{current_host} }{services} }, $text );
    }

    $self->SUPER::end_element($element);
}

```

1; # 返回真值, 保证HostHandler模块能被正常加载

快速提示: 像下面这样元素内容中出现子标签的情况, 上面的代码还无法处理:

```

<element>
  This is some text in the element.
  <sub_element> This is some text in a subelement </sub_element>
  This is some more text in the element.
</element>

```

当然, 利用XML::SAX还是可以处理这种情况的, 只不过事件处理器的复杂度会上一个台阶。这超出了我们只做基础SAX2示例的范畴。

我们差不多已经探索了基于SAX2模型读取XML文档的方法, 除此之外还有许多高级的SAX技术没法在这里一一详述。其中有一个出现在我们之前的范例代码中却一直没有解释的秘密, 那就是以\$self->SUPER::开始的行, 这里简单提一下。基于SAX2的代码可以非常便捷地构造管道式流转处理事件的工作方式, 这和Unix系统中的管道很像。当一系列SAX2事件发生时, 各个环节的事件处理器代码按自己的需求对数据进行变形或过滤, 然后再把该事件转交给下一个事件处理器接着处理。在XML::SAX里可以很方便地衔接事件处理器(而采用Barrie Slaymaker的XML::SAX::Machine模块的话会更容易)。可

以看到在上面的范例中，我们编写的方法末尾都调用了`$self->SUPER::`方法，从而保证最后执行对象在父类中的同名事件句柄，它会替我们做许多琐碎细致而且必要的常规工作。虽然你认为它表面上看好像实际派不上用场，不过最好还是例行公事地写上，有了总是不会错的。

用混合方式解析XML (XML::Twig)

XML::Twig的优缺点

优点是：

- 它提供了一种以Perl为核心诉求的解决方案。
- 它能以精巧的方式处理庞大的数据集合，节约无谓的内存和CPU开销。它特别适合于要处理的数据只是大型文档中一小部分这种情况。也就是说，你可以让XML::Twig只处理指定的元素及其子元素，它会在内存中创建这部分数据对应的对象。再要修改或取用其他元素时，会刷新这段内存，以新的数据取而代之。
- 它拥有使用类似XPath的选择器以选择要处理的数据的能力。因此可以方便地针对特定元素构建回调子例程（即给定某个XPath选择器，然后在找到符合条件元素时自动运行一段代码）。
- 该模块提供了折中的处理方式，兼顾了基于文档树处理（类似于XML::LibXML的DOM特性）和基于数据流处理（像SAX2那样）的优点。
- 也可以用它读取HTML（使用以XML方式工作的HTML::TreeBuilder，所以需要将文档完整读入内存）。
- 它提供某些输出选项，支持属性排序以及按层次结构格式化XML文档，方便阅读。
- 它宣称自己追随DWIM（Do What I Mean，按我的意思去做）的理念。
- 它有丰富详实的文档(<http://www.xmltwig.com>)，作者也提供良好的技术支持。

缺点是：

- 并非完全兼容既有标准（如XML::SAX符合SAX2、XML::LibXML实现W3C DOM模型），不过一般不会对你有影响。
- 它只实现了XPath 1.0标准的子集（但却是非常有用的子集）。
- 某些情况下性能会有所下降，速度可能比XML::LibXML还慢。
- 底层使用expat解析（可能这不是什么问题，毕竟expat已经相当稳定了，但它缺乏积极维护却是不争的事实）。

何时该用这个模块？XML::Twig特别适合于文档很大，但要处理的数据很小这种情形。一旦领会了它对整个世界（亦即像细枝嫩条般的XML文档）的思考方式，再加上一点儿XPath经验，用Perl来处理XML文档也会是一件乐事。

XML::Twig的功能和之前谈到模块的功能有相当多的重叠。和它们一样，Michel Rodriguez的XML::Twig可以在内存中创建并操作按树结构表示的XML文档（类似DOM模型），或者按照在提供基于事件的回调时解析特定的数据块。为使本节简明易懂，接下来我会关注XML::Twig独有的一些特性。没有涉及到的功能和细节，不妨移步模块官网阅读说明文档。

XML::Twig的核心思想是，XML文档理应被拆分成一堆子树（subtree）来分别处理。在附录B中我会介绍把XML文档表示为从文档根元素开始展开的一棵完整的树的记法。XML::Twig在此基础上更进一步：它允许你选择其中特定的一棵子树进行操作，而不用管其他无关的子树。“twig”本来就是枝条的意思，所以它的思路本质上就是分而治之。而对于子树的选择，可以通过XPath 1.0规范的子集来定义。在解析之前，得先提供XPath选择器，并定义好相应的回调例程（在找到匹配元素时要执行的Perl代码）。这和本章之前看到的回调例程代码类似，所不同的不过是针对物：之前是特定元素或解析事件，现在是子树。让我们通过两个简单的例子来看看它到底是如何工作的。还是用之前那份XML数据文件作为测试数据。

首先是从XML文档中提取数据的简单例子。只取出<interface></interface>元素及其内容，可以这么写：^[注17]

```
use XML::Twig;

my $twig = XML::Twig->new(
    twig_roots => {
        # 此处 $_ 被设置为当前找到的符合条件的标签元素
        'host/interface' => sub { $_->print },
    },
    pretty_print => 'indented',
);

$twig->parsefile('config.xml');
```

输出结果前半部分为：

```
<interface name="eth0" type="Ethernet">
  <arec>agatha.example.edu</arec>
```

注17：这种简单的情形可以不写任何代码，XML::Twig 提供了一个命令行工具 `xml_grep`，可以按照 `xml_grep 'host/interface' config.xml` 这样写来提取同样的元素内容。此工具还有一个基于 XML::LibXML 的版本，参见<http://xmlltwig.com/tool/>。

```

    <cname>mail.example.edu</cname>
    <addr>192.168.0.4</addr>
  </interface>
  <interface name="eth0" type="Ethernet">
    <arec>gil.example.edu</arec>
    <cname>www.example.edu</cname>
    <addr>192.168.0.5</addr>
  </interface>
  <interface name="eth0" type="Ethernet">
    <arec>baron.example.edu</arec>
    <cname>dns.example.edu</cname>
    <cname>ntp.example.edu</cname>
    <cname>ldap.example.edu</cname>
    <addr>192.168.0.6</addr>
  </interface>
...

```

这里的关键是twig_roots选项，它让XML::Twig知道我们所关心的是每个<host></host>元素内的<interface></interface>子树。对于每个匹配的分支，我们都会调用选项中给定的代码，按照漂亮的格式打印其内容。

再来看稍微复杂点的例子，要把所有的<service></service>元素全部改为<port></port>元素（并把该服务所用端口号作为属性添加进去），需要这么写：

```

use XML::Twig;
use LWP::Simple;

my %port_fix = ( 'DNS'      => 'domain',
                 'IMAP4'    => 'imap',
                 'firewall' => 'all' );
my $port_list_url = 'http://www.iana.org/assignments/port-numbers';

my %port_list = &grab_iana_list;

my $twig = XML::Twig->new(
    twig_roots => { 'host/service' => \&transform_service_tags },
    twig_print_outside_roots => 1,
);

$twig->parsefile('config.xml');

# 修改 <service> 为 <port>, 并把服务端口号作为属性加入
sub transform_service_tags {
    my ( $twig, $service_tag ) = @_;

    my $port_number = (
        $port_list{ lc $service_tag->trimmed_text }
        or $port_list{ lc $port_fix{ $service_tag->trimmed_text } }
        or $port_fix{ lc $service_tag->trimmed_text }
    );

    $service_tag->set_tag('port');
    $service_tag->set_att( number => $port_number );
}

```

```

    $twig->flush;
}

# 从其URL抓取IANA分配的端口清单,
# 并按此返回服务名和端口号映射的哈希
sub grab_iana_list {
    my $port_page = get($port_list_url);

    # 抓取出的文件每行都按此格式:
    # service      port/protocol  explanation
    # 比如:
    # http          80/tcp        World Wide Web HTTP
    my %ports = $port_page =~ /([\w-]+\s+(\d+)\s+(?:tcp|udp)/mg;

    return %ports;
}

```

让我们逐段解释。第一步，先抓取IANA分配的端口清单，然后按哈希的形式返回，以便后续查询。例子中出现的某些服务并未列在那张清单中，所以我们自己还准备了一份哈希用于辅助查询。第二步，载入XML::Twig，告诉它我们感兴趣的元素以及找到这些元素后要执行的子例程的引用。我们还设置了twig_print_outside_roots开关选项，这会让XML::Twig原封不动地输出不符合twig_roots要求的元素（而不是像前一个例子那样简单地丢弃）。一切定义妥当，便可以扣动扳机令其解析配置文件了。

解析过程一路展开，输入数据若是不符选择器要求，就照原样输出。一旦发现符合要求的，便会连同其下所有元素作为一棵子树，一齐交给预先定义好的回调子例程处理。就上面的例子来说，我们关心的是<service></service>元素及其所含文本，即服务名称。为便于查询，我们去除了服务名称中的空白字符，然后到IANA数据建立的哈希中查找对应的标准端口号。如果没有，就从我们自己定义的哈希中修正该服务的名称（比如“DNS”不是正规的服务名称，通过该表转为“domain”再查）。若仍然未果，就放弃IANA清单，直接取用自定义哈希中对应的值（比如服务“firewall”并非正规的网络服务名称，自然也不会有对应的标准端口号，所以直接取用“all”作为端口属性值）。

通过XML::Twig修改文档非常简单。用set_tag方法修改标签名，用set_att方法插入一条新的属性以指定端口号。最后一步会让XML::Twig打印修改后的子树内容并从内存中移除，以准备好后续文档内容的解析。这个flush步骤是可选的，不过它体现了XML::Twig倾向于有效利用内存的思想。一旦执行flush步骤（或者用purge清空，但不打印子树内容），有关当前子树的所有信息都将从内存中抹去，所以每个子树所占用的开销基本相同，不像基于DOM模型的表达方式那样逐个堆叠起来浪费内存。

XML::Twig提供的对象方法还有许多，Perl程序员处理XML文档也因此比以往更为轻

松。本节仅是介绍了和之前模块不同的部分，想要深入了解学习，还是需要仔细查阅说明模块文档。

至此，我们大致可以告一段落，（及至撰写本书之时）三种最为常见的解析XML的方式已经一一呈现。现在你的工具箱里已经有了各式长刀短枪，什么情况该用什么武器，我想你应该已经有点概念了吧。

作为本节最后一个提示，我想提醒你，除此之外还有许多新生模块正在不断涌现和成长，我们应该关注它们的变化，看看它们是否趋于成熟。有两个模块我觉得挺有意思的，可推荐你去看看，一个是Jenda Krynicky的XML::Rules，另一个是Mark Overmeer的XML::Compile。

有关XML的处理方式都已讲完，怎么样，仍然不觉得它是你目前选作配置文件格式的最佳选择？好吧，我们继续。

YAML

有些人认为XML太啰嗦，每块内容都要用首尾标签圈起来，尖括号这种东西少点就好了。如果你也这么认为，倒是可以试试轻量级的格式，YAML（首字母取自YAML Ain't Markup Language，这是一种用自己的名字递归解释含义的缩写）。它要解决的问题和XML有些许不同，不过作为配置文件格式，还是可圈可点。

YAML 尝试在结构标记的复杂度和字段语义的精确度之间取得某种平衡，所以从形式上来看，YAML清爽不少。下面是从之前的XML配置文件转换而来的YAML配置文件：

```
---
network:
  description:
    name: Boston
    text: This is the configuration of our network in the Boston office.
  hosts:
    - name: agatha
      os: linux
      type: server
      interface:
        - name: eth0
          type: Ethernet
          addr: 192.168.0.4
          arec: agatha.example.edu
          cname:
            - mail.example.edu
      service:
        - SMTP
        - POP3
        - IMAP4
```

- name: gil
 - os: linux
 - type: server
 - interface:
 - name: eth0
 - type: Ethernet
 - addr: 192.168.0.5
 - arec: gil.example.edu
 - cname:
 - www.example.edu
 - service:
 - HTTP
 - HTTPS

- name: baron
 - os: linux
 - type: server
 - interface:
 - name: eth0
 - type: Ethernet
 - addr: 192.168.0.6
 - arec: baron.example.edu
 - cname:
 - dns.example.edu
 - ntp.example.edu
 - ldap.example.edu
 - service:
 - DNS
 - NTP
 - LDAP
 - LDAPS

- name: mr-tock
 - os: openbsd
 - type: server
 - interface:
 - name: fxp0
 - type: Ethernet
 - addr: 192.168.0.1
 - arec: mr-tock.example.edu
 - cname:
 - fw.example.edu
 - service:
 - firewall

- name: kersp
 - os: osx
 - type: client
 - interface:
 - name: en0
 - type: Ethernet
 - addr: 192.168.0.100
 - arec: kersp.example.edu
 - name: en1


```

    type: AirPort
    addr: 192.168.100.100
    arec: krosp.wireless.example.edu

- name: zeetha
  os: osx
  type: client
  interface:
    - name: en0
      type: Ethernet
      addr: 192.168.0.101
      arec: zeetha.example.edu
    - name: en1
      addr: 192.168.100.101
      type: AirPort
      arec: zeetha.wireless.example.edu

```

看起来非常直观，层次关系、字段名称和内容都清清楚楚。不过这只是字面上转换而来的结果，所以尽可能地保留了原来XML中的属性名称（而YAML本身并没有标签属性这种概念，所以每个原始的属性名称和值都用类似哈希的方式罗列）。如果实际应用中并不要求即时的格式转换，那按照配置文件的内容语义重新编排一份配置文件应该更为妥当。比如，下面是之前讲到XML::Parser时输出的YAML格式的结果：

```

agatha:
  interfaces:
    eth0:
      addr: 192.168.0.4
      arec: agatha.example.edu
      cnames:
        - mail.example.edu
      type: Ethernet
  os: linux
  services:
    - SMTP
    - POP3
    - IMAP4
  type: server
...

zeetha:
  interfaces:
    en0:
      addr: 192.168.0.101
      arec: zeetha.example.edu
      type: Ethernet
    en1:
      addr: 192.168.100.101
      arec: zeetha.wireless.example.edu
      type: AirPort
  os: osx
  type: client

```

本质上没太大不同，只不过去除了无谓的标签之后，内容间的层次关系更加简洁清晰。

用于解析YAML^[注18]的Perl模块，直接叫做YAML，基本用法如下：

```
use YAML qw(DumpFile); # 在该模块内部查找并加载合适的YAML解析处理模块
my $config = YAML::LoadFile('config.yml');

# （稍后）将配置数据写回文件
YAML::DumpFile( 'config.yml' , $config );
```

YAML模块本身只是其他YAML解析器的前端，这一点和XML::SAX相似。默认情况下它仅提供简单的Load/Dump过程调用，只能操作内存中的数据，但也可以用LoadFile和DumpFile直接从文件读取再写回文件。基本上也就这两种操作了：要么从什么地方Load YAML数据，要么把数据Dump为YAML格式的内容。

要是希望用面向对象的方式操作YAML，可以选择Config::YAML模块。另外还有一个速度极快的YAML解析/输出模块YAML::XS，它是基于libyaml库实现的。如果不强求纯Perl实现的话，推荐使用该模块（如果可用的话，YAML会默认在后端尝试委托此模块工作）。

最后介绍的配置文件格式虽然简单，但功能却很强大。本章到此差不多也该结束了。当然，除了我们这里谈到的配置文件格式以外，还有许多甚至可以说无限种其他配置文件格式，但最常见最有用的也就是这些了。

多功能合一模块

以上我们介绍了正确解析配置文件该用的模块。在结束本章之前，我还想快速简单提一下其他几个有用的模块。

Michael Graham的Config::Context模块是对Config::General、XML::Simple以及Config::Scoped的封装。只要是这三个模块所支持的配置文件格式，都可以通过Config::Context的单一接口处理。因此，它自然地支持（像Apache配置文件格式那样的）上下文配置，比如可以在配置文件中用<Location></Location>标签定义局部配置变量。

要是你热切期望能有一个万能模块来支持绝大多数配置文件格式，可以看看Jos Boumans

注18：YAML 有个很棒的优点，它可以独立于任何一种编程语言存在。几乎每种语言都提供了YAML 解析器和生成工具，如Ruby、Python、PHP、Java、OCaml，甚至 JavaScript 也是。所以YAML 可用于不同语言间交换复杂数据。

的`Config::Auto`。它能处理以逗号、空格或等号分隔的键-值对格式、XML格式、Perl代码、`.ini`格式、以及BIND9风格和`irssi`格式等。不仅如此，它（默认）还会自动猜测配置文件的格式，无需预先指定。当然，如果不放心，也可以说明按指定的格式解析处理。

高级配置信息存储机制

说了那么多有关配置文件的话题，你大概有点厌烦了吧（我很理解你），那最后让我再简单提一下高级配置信息存储来结束本章。除了普通的配置文件之外，实际上还有许多更为自然合理的地方可用于存储配置信息。^[注19]要是性能要求极为严苛，可以考虑使用共享内存区段的方式使用配置信息。另外，当今有许多系统都将自己的配置信息通过DBI（参见第7章）存放于数据库中。也有的系统会通过特定的网络服务器分布配置信息。这些尝试都非常有意思，值得探索一番，不过已经超越本书范围了，就不多说了。

本章所用模块

模块名	CPAN ID	版本
Readonly	ROODE	1.03
Storable (随 Perl 发布)	AMS	2.15
DBM::Deep	RKINYON	1.0013
Text::CSV::Simple	TMTM	1.00
Text::CSV_XS	JWIED	0.23
Config::Std	DCONWAY	0.0.4
Config::General	TLINDEN	2.31
Config::Scoped	GAISSMA	0.11
Config::Grammar	DSCHWEI	1.02
XML::Writer	JOSEPHW	0.606
XML::Simple	GRANTM	2.18
XML::LibXML	PAJAS	1.69
XML::SAX	GRANTM	0.96

注19：当然，也有许多不那么合乎情理的地方，比如，通过 `Acme::Steganography::Image::Png` 在图片文件中写入隐藏信息，或是用 `Acme::Playwright` 在剧本中嵌入等等。

模块名	CPAN ID	版本
XML::Parser	MSERGEANT	2.36
XML::Twig	MIROD	3.32
LWP::Simple (随 Perl 发布)	GAAS	5.810
YAML	INGY	0.68
Config::YAML	MDXI	1.42
YAML::XS	NUFFIN	0.29
Config::Context	MGRAHAM	0.10
Config::Auto	KANE	0.16

更多参考资料

本章中的有些材料来自原本给USENIX协会《login》杂志2006年2月刊写的专栏，做了改动和修订。

Damian Conway的《Perl Best Practices》(O'Reilly)一书中也有一节很好的关于配置文件的内容。

XML和YAML

<http://msdn.microsoft.com/xml>和<http://www.ibm.com/developer/xml>都包含了丰富的信息。Microsoft和IBM对XML还是非常上心的。

http://www.activestate.com/support/mailling_lists.htm上维护着Perl-XML邮件列表。它（及其归档资料）是关于本章主题最好的参考资源之一。

<http://www.w3.org/TR/1998/REC-xml-19980210>是事实上的XML 1.0规范说明书。只要是用XML的人，最后都难免要阅读这份完整的规范说明说，但如果你只想快速查阅参考手册，我建议你阅读接下来两处将要介绍的带注释的版本。

<http://www.xml.com>上有很多很好的作为参考资料的文章和XML链接。它同时还提供了一个很棒的由Tim Bray（XML规范作者之一）创建的带注释版本的XML规范说明说。

《XML: The Annotated Specification》由Bob DuCharme编著（Prentice Hall出版），是另一份极好的带注释的规范说明书版本，包含了大量的XML代码范例。

《XML Pocket Reference》(Third Edition)，由St.Laurent和Fitzgerald合著（O'Reilly出版），它是一本简明但综合性非常强的XML介绍书籍，适合那些没有耐心的读者。

《Learning XML》(Second Edition), 由Erik T. Ray所著 (O'Reilly出版), 以及《Essential XML: Beyond Markup》, 由Don Box等人合著 (Addison-Wesley出版), 也是不错的学习一系列基于XML的技术的书籍, 包括XPath。后者虽然和Perl关联比较少, 但内容更详实, 所涉及的深度是我在其他参考书籍中从未见过的。

《Perl and XML》, 由Erik T. Ray和Jason McIntosh合著 (O'Reilly出版), 也很值得一看, 尽管它是基于那个时期的老XML模块的。自2002年发布以来, Perl的XML领域已经有了不少变化, 但这本书对于仍在用老模块的人来说不失为一个很好的参考。

<http://perl-xml.sourceforge.net>是Perl XML相关开发信息的集中站点。该站点上的FAQ和Perl SAX相关页面对你来说应该是非常重要的阅读资料。

<http://xmlsoft.org>是Gnome的libxml库的官方站点, XML::LibXML便是基于该模块。若要弄懂XML::LibXML模块中某些晦涩部分, 你总有一天会找到这里的。

<http://www.saxproject.org>是SAX2的官方站点。

《Object Oriented Perl: A Comprehensive Guide to Concepts and Programming Techniques》, 由Damian Conway编著 (Manning出版), 是学习Perl面向对象编程最好的资料。理解Perl的面向对象编程对使用好XML::SAX模块是非常重要的。

<http://www.xmltwig.com>是XML::Twig模块的官方站点, 并且这里还有丰富的好文档、教程、幻灯片等资料。

<http://www.yaml.org>是所有关于YAML内容的站点。

SQL数据库管理

怎么会在系统管理的书里面加入一章数据库管理的内容呢？其实系统管理员以及对Perl感兴趣的人往往同样需要了解数据库，原因如下：

- 其实只要你大致看看这本书的前几章，就会发现数据库对于现今系统管理的重要性。我们之前用了简易数据库来保存用户和主机信息，但那只是冰山一角。邮件列表、密码文件还有Windows注册表都可以说是日常使用的数据库。所有大型系统管理软件包（比如CA、Tivoli、HP或者微软的产品）都依赖于数据库后端。如果你想进一步提升系统管理水平，那么迟早会涉足数据库领域。
- 数据库管理可以说是迷你的系统管理。数据库管理员(DBA)要考虑的问题包括：
 - 登录/用户
 - 日志文件
 - 存储管理（比如空间管理等）
 - 进程管理
 - 连通问题
 - 备份
 - 安全/基于角色的访问控制（RBAC）

听上去很常见？我们应该可以融会贯通这两个领域的知识。

- Perl是一种胶水语言，可能是迄今为止最好的。目前已经为Perl数据库连接方面做了很多工作，相信主要动力来自于Web开发项目。我们可以充分利用这个成果。尽管Perl已经能处理多种不同数据库格式（比如 Unix DBM、Berkeley DB等等），但这一章我们会专注于大型数据库。当然在本书其他部分，我们还会继续关注其他格式的数据库。

- 我们使用的大多数应用需要把数据存储数据库中，比如Bugzilla、Request Tracker或者网络日历等。为了很好地理解并支持这些软件的运行，我们需要调查它们背后的数据库，从而确保信息存取的效率。
- 还有一个原因就是数据库存储了有用的信息。这听上去有些莫名其妙，但确实其中有些信息对我们是有用的：日志、性能指标（比如趋势分析或者计算能力规划）、关于用户和系统的元信息等等。

要成为一个懂得数据库的系统管理员，你必须了解一些SQL（Structured Query Language，结构化查询语言），这是多数商业（或某些非商业）数据库的“通用语”（Lingua franca）。编写数据库管理使用的Perl脚本需要了解一些SQL，因为大多数程序需要嵌入SQL语句。可以参考附录D来学习最基础的SQL语句。为了避免偏题，这一章的数据集将和前面章节保持一致。

从Perl中与SQL服务器交互

在过去的某个阶段，曾经有很多Perl模块是用来与各种数据库交互的。每次想要连接某个厂商的数据库之前，都得先找到合适的模块，并且学习模块的使用方法。如果你在项目的某个阶段切换数据库，那么你可能要重写所有的数据存取代码。所以Tim Bunce开发了DBI（DataBase Interface）模块，这使得Perl爱好者大为受益。

DBI可以被称为是一个“中间件”，它的存在使得程序员可以专注于编写高层的DBI调用，不必学习每种数据库的特殊API。DBI模块可以通过DBD（DataBase Dependent）驱动，把高层调用翻译成底层的数据库API调用。正是这个特定数据库驱动程序替我们处理了每种数据库特殊的通信方式。

其实正是因为这个卓越的想法，我们才有了其他语言中见到的类似概念（比如JDBC），甚至连Windows操作系统内置的ODBC（Open DataBase Connectivity，开放数据库连接）也是对它的效仿。其实ODBC并非DBI的竞争对手，因为它在Windows世界非常普及，所以我们会花些篇幅介绍它。Windows的Perl程序员都非常熟悉ODBC数据源，所以我们会对它们做些简单对比。然而这个问题对非Windows的技术人员也有帮助，因为到处都有“杂货铺”类型数据库需要类似ODBC这样的编程方法。

图7-1展示了DBI和ODBC的架构。两者都（至少）有三层结构：

1. 底层数据库，如Oracle、MySQL、Sybase或者Microsoft SQL Server等等。
2. 特定数据库层，用来代替程序员处理与数据库服务器的交互。程序员不必直接与这一层打交道，往往是使用第三层。对DBI来说，都是通过某个特殊的DBD模块来实际处理这一层。比如对于Oracle数据库来说，DBD::Oracle模块会被调用。DBD模

块通常在服务器厂商提供的特定服务器客户端库建立之后才能被链接。对ODBC来说，厂商提供的特定数据源ODBC驱动程序在这一层工作。

3. 数据库无关接口层。很快我们就会在这一层写Perl脚本。在DBI中，这一层被称为DBI层，因为这层主要是做些DBI调用。在ODBC中，这层的程序通过ODBC API调用与ODBC驱动程序管理器交互。

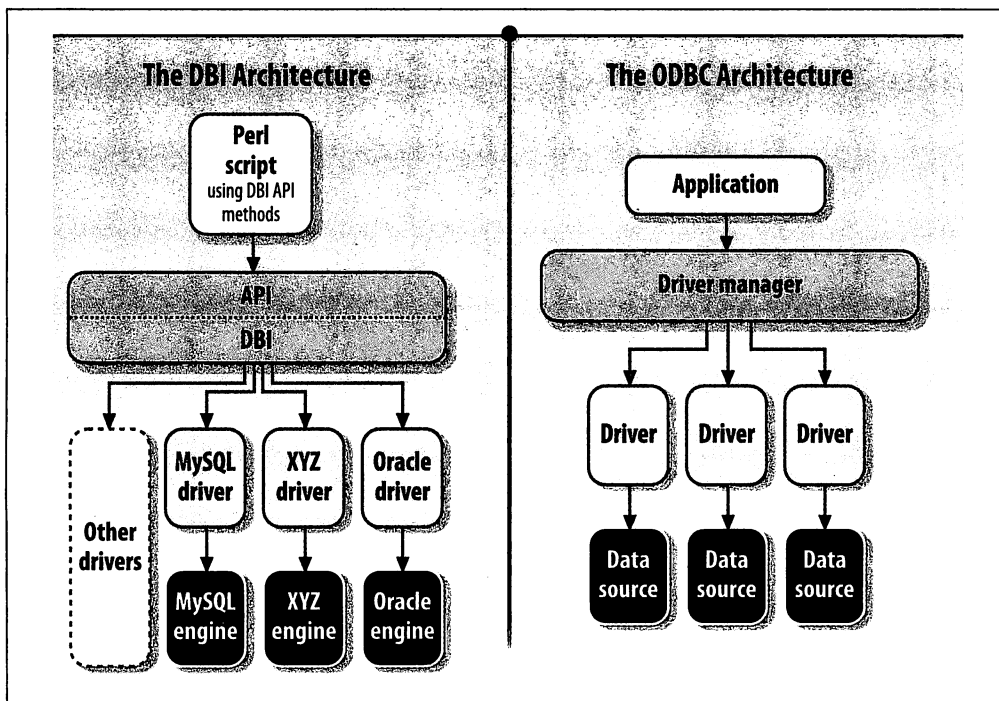


图7-1：DBI和ODBC架构

这个模型的优点是，为DBI或ODBC开发的程序大多具有很好的（跨服务器、跨厂商）可移植性。API调用是一致的，与底层数据库无关（最起码理论上如此）。不幸的是，我们将要编写的数据库管理代码几乎总是与平台相关，因为几乎没有两种服务器会采用相同的管理架构。^[注1]有经验的系统管理员喜爱可移植性，但并不总是期待代码可移植。

有了这些背景知识，我们就可以尝试尽快写点代码了。与DBI打交道的方式非常简单，因为只有一个DBI模块。ODBC则要复杂些，因为有两种和ODBC打交道的方式。一度Win32::ODBC是主要的渠道，但最近DBI框架的DBD模块DBD::ODBC逐渐成为了主流的方法（甚至Win32::ODBC的作者也开始推荐它）。DBD::ODBC把ODBC世界统一到DBI的下面，使ODBC成为一种特殊的数据源。我们很快就会介绍相关例子。

注1：Microsoft SQL Server 是从 Sybase 的源代码衍生出来的，这是一个罕见的反例。

对DBI范例代码来说，我们会使用MySQL和Oracle服务器，而对ODBC来说，我们会使用Microsoft SQL Server。

从Unix中访问Microsoft SQL Server

跨平台系统管理员常常会面对一个问题：“如何从Unix主机访问Microsoft SQL Server呢？”如果某个机构主要的管理和监控系统是基于Unix的，那么新安装的Microsoft SQL Server就成为了一个挑战。我起码知道解决这个问题的四种方法。其中方法2和3是与Microsoft SQL Server并不直接相关的，所以你可以在其他数据库系统中应用。可选方案有：

1. 安装并使用DBD::Sybase模块。这个模块依赖于底层的数据库通信库，目前有两组库可供选择。第一个是用Sybase OpenClient库，这个库可能在你的平台上能够工作（比如Linux版本的Sybase Adaptive Server Enterprise就带有这个免费的库）。第二个选择是安装<http://www.freetds.org>提供的FreeTDS库。可以参考站点上的信息来安装合适的版本，以配合你的服务器版本。
2. 使用一种“代理”驱动。目前DBI模块自带了两个DBD代理模块：比较老的称为DBD::Proxy，比较新的称为DBD::Gofer。两者都允许你连接到SQL Server服务器上运行一个小的网络取务来透明地代理从你的Unix客户机至服务器的请求。
3. 安装并通过DBD::ODBC使用Unix ODBC软件。另外还需要从MERANT(<http://www.merant.com>)或者OpenLink Software(<http://www.openlink.com>)处采购驱动程序，当然也可以使用开源驱动程序。可以参考iODBC(<http://www.iodbc.org>)和unixODBC(<http://www.unixodbc.org>)的主页。总之你需要有Unix平台的ODBC驱动程序（一般是数据库厂商提供的）以及ODBC管理程序（如unixODBC或者iODBC）。
4. Microsoft SQL Server 2000以上的版本支持通过HTTP或HTTPS来监听数据库查询请求，当然这还需要配置一台Web服务器（例如IIS）。查询结果以XML格式返回，这样的格式易于处理，可以参考第6章中的例子。

使用DBI框架

下面是使用DBI^[注2]的基础步骤：

1. 载入必要的Perl模块。

注2：要了解更多关于DBI的信息，请参考《Programming the Perl DBI》，作者是Alligator Descartes和Tim Bunce（O'Reilly出版）。

其实也没什么特殊的，我们需要的只是这一行：

```
use DBI;
```

2. 连接数据库并获得一个连接句柄。

使用Perl DBI来连接MySQL数据库并返回数据库句柄的代码看起来如下：

```
# 连接到名为 $database的数据库，并使用给定的
# 用户名和密码访问，然后返回数据库句柄
my $database = 'sysadm';
my $dbh = DBI->connect("DBI:mysql:$database",$username,$pw);
die "Unable to connect: $DBI::errstr\n" unless (defined $dbh);
```

DBI会载入底层的DBD驱动程序（DBD:mysql），以便连接到服务器。然后测试connect()是否成功。DBI为connect()提供了RaiseError和PrintError选项，这样我们可以让DBI对会话中所有DBI操作的返回代码进行测试并在错误发生时投诉。比如我们可以使用下面的代码：

```
$dbh = DBI->connect("DBI:mysql:$database",
                    $username,$pw,{RaiseError => 1});
```

这里DBI如果发现connect()失败会自动调用die。

3. 发送SQL命令给服务器。

有了载入的Perl模块和活跃的数据库连接，我们就可以大展身手了！我们先尝试给服务器发送一些SQL命令。我们会使用附录D里面的SQL查询作为例子。这些查询语句会使用Perl的q惯例来引起（也就是使用q{something}来表示something），这样我们就不必担心查询中嵌入的单引号或者双引号了。下面就是DBI发送命令用的两个方法之一：

```
my $results=$dbh->do(q{UPDATE hosts
                        SET bldg = 'Main'
                        WHERE name = 'bendir'});
die "Unable to perform update:$DBI::errstr\n" unless (defined $results);
```

\$results里面会包含修改过的记录行数，或者是undef，但这意味着发生了问题。尽管我们知道了修改的行数，但还是不如实际执行一条SELECT语句来得清楚。下面展示的第二个方法能让我们看到实际的数据。

在使用第二个方法之前，你需要先prepare SQL语句，然后让服务器execute它。例子如下：

```
my $sth = $dbh->prepare(q{SELECT * from hosts}) or
die 'Unable to prep our query:'.$dbh->errstr."\n";
my $rc = $sth->execute or
die 'Unable to execute our query:'.$dbh->errstr."\n";
```

`prepare()`返回的对象我们还没有见过，它被称为语句句柄（statement handle）。就像数据库句柄代表活跃的数据库连接那样，语句句柄代表了已经`prepare`的语句。一旦拥有这个句柄，我们就可以用`execute`来把查询实际发送到服务器。之后我们还会使用这个句柄来获取查询结果。

你可能会奇怪为什么先要`prepare()`一条语句，而不是直接执行它。`prepare()`给了DBD驱动程序（其实是底层的数据库客户端库程序）一个机会来分析SQL查询。一旦语句被`prepare()`之后，我们可以反复执行，不必再分析它（也就不必再反复确定服务器端的执行计划）。通常这能节约一些服务器资源。实际上哪怕是使用DBI默认的`do()`方法执行语句，也会在幕后默认执行`prepare()`和`execute()`。

如同之前看到的`do`调用那样，`execute()`方法也会返回被影响的行数。如果查询没有影响任何一行，那么会返回0E0字符串，这个字符串作为数值时是零，但对Perl来说却是真。如果驱动程序认为无法确定被影响的行数，那么会返回-1。

在我们进一步分析取回查询结果的方法之前，先花点时间讨论一个大多数DBD模块都支持的`prepare()`特性：占位符（placeholder），也称为位置标记（positional marker）。它让`prepare()`SQL语句时在语句中留下的空缺在`execute()`时得到填充。这通常能让我们写出节约分析时间的SQL语句。在这里我们使用问号（?）来作为标量的占位符。下面的代码展示了我们如何使用占位符：^[注3]

```
my @machines = qw(bendir shimmer sander);
my $sth = $dbh->prepare(q{SELECT name, ipaddr FROM hosts WHERE name = ?});
foreach my $name (@machines){
    $sth->execute($name);
    do-something-with-the-results
}
```

在`foreach`循环的每次迭代中，`SELECT`查询都会执行不同的`WHERE`语句。同样，多个占位符也不难理解：

```
$sth->prepare(
    q{SELECT name, ipaddr FROM hosts
    WHERE (name = ? AND bldg = ? AND dept = ?)});
$sth->execute($name,$bldg,$dept);
```

使用占位符的另一个好处是自动引起参数。

现在我们知道了如何获取非`SELECT`查询影响的行数，那么如何获取`SELECT`查询的结果呢？

4. 获取SELECT结果。

注3： 以上展示了最常见的案例，也就是用字符串代替查询中的占位符。如果你想用更复杂的数据类型作为占位符（比如日期类型），那么你需要使用DBI的`bind_param()`方法来绑定，然后才能调用`execute()`。

DBI提供了三种不同的方法来获取查询结果。我们会逐个展示它们，因为在多变的需求中，它们可能有各自的用处。

其中两个机制有些类似于在附录D中我们所讨论的游标。这个机制让我们逐行遍历查询结果，并调用某个方法让下一行结果出现。

第一个机制使用`bind_col()`或者`bind_columns()`，结合`fetchrow_arrayref()`。这往往是最佳搭配，因为它是最高效的，也是最有魔力的机制。让我们看看它是如何工作的。在`execute()`之后，我们让DBI把结果放在我们选择的标量或者标量的集合（也就是列表或者哈希）中。查询结果和变量之间的绑定是这样建立的：

```
# 假设我们刚刚完成这条语句的查询：SELECT first,second,third FROM table
my $first;
my $second;
my $third;
$sth->bind_col(1, \ $first); # 将查询结果的第一列绑定到 $first
$sth->bind_col(2, \ $second); # 绑定第二列
$sth->bind_col(3, \ $third); # 绑定第三列，以此类推

# or perform all of the binds in one shot:
$sth->bind_columns(\ $first, \ $second, \ $third);
```

绑定到整个数组或者哈希元素的语法也大致相似，都是使用`\(...)`：

```
$sth->bind_columns( \@array ); # $array[0] 对应于第一列的值
                                # $array[1] 对应于第二列的值……

# 我们只能绑定到哈希元素，无法直接绑定到哈希本身
$sth->bind_col(1, \ $hash{first} );
$sth->bind_col(2, \ $hash{second} );
```

现在我们每次调用`fetch()`之后，这些变量都会自动获取查询结果下一行的数据：

```
while ($sth->fetch){
    # 某些操作作用 $first、$second 和 $third 的代码
    # 或者用 $array[0], $array[1]……
    # 又或者用 $hash{first}, $hash{second}
}
```

其实`fetch()`只是`fetchrow_arrayref()`方法的别名，这让我们自然转到下一个DBI获取SELECT结果的机制。如果你觉得绑定列的方式有些太魔幻，或者你喜欢结果以某种Perl数据结构的方式获取，那么可以从多种方法中选择其一调用。

在DBI中，我们可以用表7-1中的一种方法来获取查询结果。

表7-1：DBI获取数据的方法

方法名	返回数据	无更多数据时返回
fetchrow_arrayref()	以数组引用的格式返回匿名数组， 其中存放了结果集中下一行的所有 字段值	undef
fetchrow_array()	返回一个数组，其中的值为结果 集中下一行的所有字段值	空列表
fetchrow_hashref()	返回一个哈希引用，指向匿名哈 希，哈希的键是各个字段名，值 则是结果集中下一行各个字段的值	undef
fetchall_arrayref()	返回一个数据结构（数组的数组） 的引用	一个空数组的引用
fetchall_hashref(\$key_field)	返回一个数据结构（哈希的哈希） 的引用。外层哈希的键是\$key_ field字段的唯一返回值，而内部的哈 希则类似于fetchrow_hashref()返 回的结构	一个空哈希的引用

有两个方法是返回列表的：fetchrow_这类单行结果方法，以及fetchall_这类多行结果方法。fetchrow_方法会把当前行的结果返回，类似于我们之前看到的方法。而fetchall_则会把整个结果集返回（相当于反复调用fetchrow_直到数据取完）。要非常小心地使用这个方法，因为它可以把非常大的结果集放入内存。如果你的结果集达到TB级别，那么这个方法肯定要出问题。

让我们分别查看每个方法的使用。对下面列出的每个方法，我们假定已经执行以下代码：

```
$sth = $dbh->prepare(q{SELECT name,ipaddr,dept from hosts}) or  
    die 'Unable to prepare our query: '.$dbh->errstr."\n";  
$sth->execute or die "Unable to execute our query: ".$dbh->errstr."\n";
```

这是作用中的 fetchrow_arrayref()：

```
while (my $aref = $sth->fetchrow_arrayref){  
    print 'name: ' . $aref->[0] . "\n";  
    print 'ipaddr: ' . $aref->[1] . "\n";  
    print 'dept: ' . $aref->[2] . "\n";  
}
```

警告：出于善意提醒`fetchrow_arrayref()`的用户：如果你开始依赖于数组中元素的顺序（也就是说第几个字段应该是什么内容），那么这样的代码迟早要让你吃苦头。只要有人无意中修改了之前的`SELECT`语句，那么没人能知道`$aref->[2]`里面到底会是什么内容。

DBI说明文档提到`fetchrow_hashref()`比`fetchrow_arrayref()`要慢一些，因为有更多的处理步骤，但这个代价是值得的，它能产生更加可读以及可能更加可维护的代码。看看下面的例子：

```
while (my $href = $sth->fetchrow_hashref){
    print 'name: ' . $href->{name} . "\n";
    print 'ipaddr: ' . $href->{ipaddr} . "\n";
    print 'dept: ' . $href->{dept} . "\n";
}
```

最后让我们看看`fetchall_arrayref()`。返回的每个引用看上去都与`fetchrow_arrayref()`返回的结果相似，如同图7-2所示。

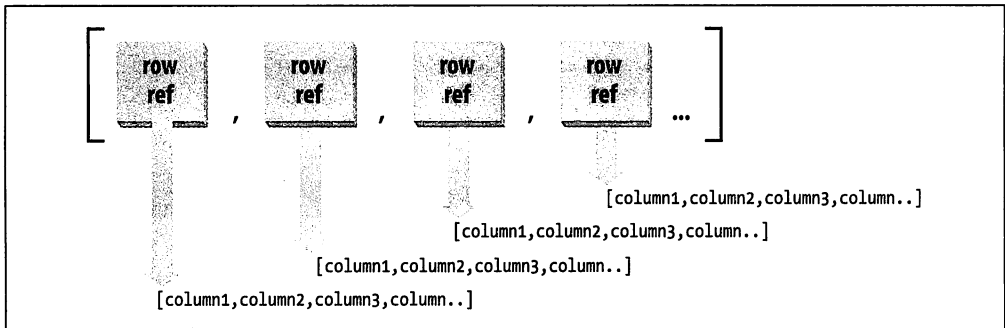


图7-2: `fetchrow_arrayref()` 返回的数据结构

下面的代码能打印整个查询结果集：

```
$aref_aref = $sth->fetchall_arrayref;
foreach my $rowref (@$aref_aref){
    print 'name: ' . $rowref->[0] . "\n";
    print 'ipaddr: ' . $rowref->[1] . "\n";
    print 'dept: ' . $rowref->[2] . "\n";
    print '-' x 30, "\n";
}
```

这段代码仅限于我们的查询结果集，因为它依赖于每个字段在数组中的顺序。例如我们假定主机名总是在结果数组的第一个位置（`$rowref->[0]`）。

我们可以使用SQL语句句柄的某些特殊属性（有时称为元数据）来重写以上代码，使它更加通用。如果我们在执行查询之后查看`$sth->{NUM_OF_FIELDS}`，那么它会

告诉我们查询结果集的字段数量。而`$sth->{NAME}`则会用数组引用的方式告诉我们每个结果字段的字段名。所以下面的代码更加通用一些：

```
my $aref_aref = $sth->fetchall_arrayref;
my $numfields = $sth->{NUM_OF_FIELDS};
foreach my $rowref (@$aref_aref){
    for (my $i=0; $i < $numfields; $i++){
        print $sth->{NAME}->[$i].": ".$rowref->[$i]."\n";
    }
    print '-'x30,"\n";
}
```

请查看DBI文档中更多关于元数据属性的信息。

最后在表7-2中还列出了一些“快捷”方法，它们能把SQL语句的分析、执行和返回的数据结合起来。

表7-2：DBI快捷方法

方法名	这个方法结合的那些方法
<code>selectcol_arrayref(\$stmtnt)</code>	<code>prepare(\$stmtnt)</code> 、 <code>execute()</code> 和 <code>(@{fetchrow_arrayref()})[0]</code> （也就是返回所有行的第一列，当然可以用可选参数 <code>Columns</code> 来设定列编号的列表）
<code>selectrow_array(\$stmtnt)</code>	<code>prepare(\$stmtnt)</code> ， <code>execute()</code> ， <code>fetchrow_array()</code>
<code>selectrow_arrayref(\$stmtnt)</code>	<code>prepare(\$stmtnt)</code> ， <code>execute()</code> ， <code>fetchrow_arrayref()</code>
<code>selectrow_hashref(\$stmtnt)</code>	<code>prepare(\$stmtnt)</code> ， <code>execute()</code> ， <code>fetchrow_hashref()</code>
<code>selectall_arrayref(\$stmtnt)</code>	<code>prepare(\$stmtnt)</code> ， <code>execute()</code> ， <code>fetchall_arrayref()</code>
<code>selectall_hashref(\$stmtnt)</code>	<code>prepare(\$stmtnt)</code> ， <code>execute()</code> ， <code>fetchall_hashref()</code>

5. 关闭与数据库的连接。

对 DBI 来说，这很容易：

```
# 从数据库断开连接
$dbh->disconnect;
```

从DBI中使用ODBC

其实在DBI中使用ODBC与之前介绍的步骤并没什么本质区别，只有一个小小的例外。主要问题来自于初始的`connect()`调用，我们要使用正确的参数。ODBC要求我们在连接之前先完成数据源名（Data Source Name, DSN）的配置。DSN的背后是一些连接数据

库用的配置信息（比如服务器名、数据库名等等）。DSN有两种选择：用户DSN和系统DSN，主要的区别是这个配置信息的共享范围是创建的用户还是机器上的所有用户。^[注4]

DSN既可以通过Windows的ODBC控制面板创建（参考图7-3），也可以通过Perl编程创建。

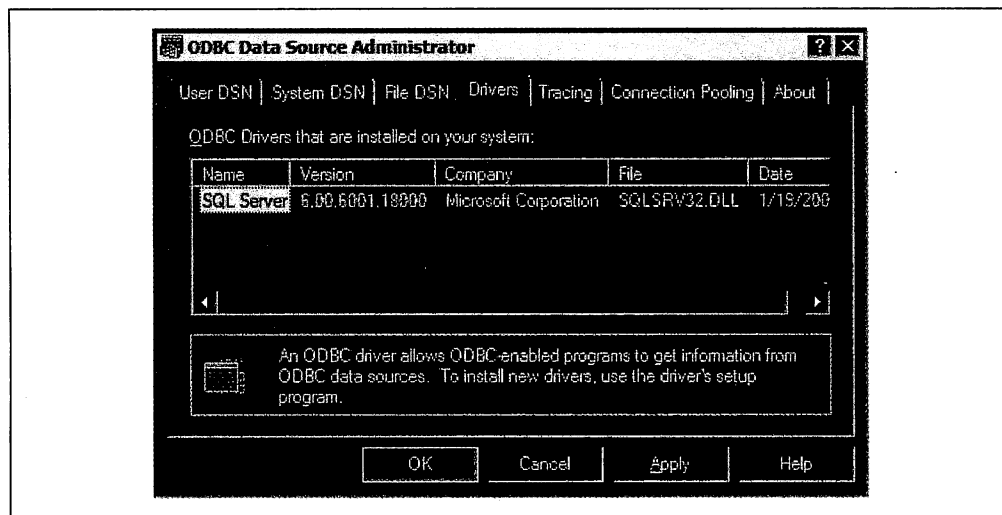


图7-3：Windows ODBC控制面板

我们会尝试后者，为了满足Unix爱好者对命令行的偏爱（当然后面的“注意”还会列出更好的原因）。下面的代码能创建SQL Server数据库的用户DSN：

```
use Win32::ODBC; # 我们仅使用该模块创建DSN；
                  # 而其他工作都将通过DBI交给DBD::ODBC处理

# 创建一个用户DSN，连接到Microsoft SQL Server
# 注意：如果要创建系统DSN，把ODBC_ADD_DSN替换为ODBC_ADD_SYS_DSN即可，
# 但请确定仅当程序需要给其他用户运行时才使用系统级别的DSN
# （比如在Web应用中）
#
if (Win32::ODBC::ConfigDSN(
    ODBC_ADD_DSN,
    'SQL Server',
    ('DSN=PerlSysAdm',
     'DESCRIPTION=DSN for PerlSysAdm',
     'SERVER=mssql.example.edu', # 服务器名称
     'ADDRESS=192.168.1.4',      # 服务器 IP 地址
     'DATABASE=sysadm',         # 我们要访问的数据库
     'NETWORK=DBMSSOCN',        # TCP/IP Socket 库
```

注4：其实还有第三种选择：文件DSN。可以把DSN配置信息存储在文件中，这样多台机器之间可以共享，但我们将要使用的Win32::ODBC并不支持这个。


```
    ))){\n        print "DSN created\\n";\n    }\n    else {\n        die "Unable to create DSN:" . Win32::ODBC::Error( ) . "\\n";\n    }\n}
```

注意：到底是应该手动创建DSN还是应该自动生成呢？这个问题并没有绝对正确的答案。一方面DSN是对如何存取关键和敏感数据的描述。所以说是谁创建并测试此DSN是非常关键的，另外如何测试也非常关键（我建议手动测试更加稳妥）。如果我们刻意从机器上删除某个DSN，但它又立刻被自动重建，那往往会出问题。另一方面，手动创建很容易出错，并且在某些需要批量创建的场合并不容易操作。

最好的答案应该是编写并测试一系列配置脚本，然后既可以手动运行，也可以自动配置。这应该能解决这个两难的问题。

一旦有了可用的DSN，你就可以在`connect()`调用中使用它。比如我们可以用如下代码连接之前创建的DSN：

```
use DBI;\n\n$dbh = DBI->connect('DBI:ODBC:PerlSysAdm',$username,$pw);\ndie "Unable to connect: $DBI::errstr\\n" unless (defined $dbh);
```

从这一刻开始，你的那些DBI技巧全都可以奏效了。DBD::ODBC的说明文档还列出了此模块提供的一些其他功能以及使用时需要注意的问题。你现在已经知道了如何使用DBI和ODBC从Perl来访问数据库，所以下面我们会把注意力放在如何用Perl进行数据库管理上。

服务器文档化

如果我们已经花了很多时间和精力来配置SQL服务器和其中存储的资料，那么保持一份最新的数据库文档会非常有益。如果数据库崩溃了并且没有找到备份，你可能会被要求重建所有的表。你也可能需要把数据从一台服务器迁移到另一台，那么了解原始库和目标库的数据结构就显得非常重要。哪怕只是你自己的数据库编程，能看到表结构也会非常有用。

为了让你了解数据库管理的不可移植特性，我会展示为此任务编写的三种不同的数据库代码（通过DBI和ODBC，包括Win32::ODBC）。这三数据库程序完成的共同任务是：打印服务器上所有的数据库，罗列其中的表和表结构。这个脚本可以很容易地扩充出更多功能，比如列出表中那些可以为空（或者不能为空）的字段。大体上程序的输出如下所示：

```

---sysadm---
    hosts
        name [char(30)]
        ipaddr [char(15)]
        aliases [char(50)]
        owner [char(40)]
        dept [char(15)]
        bldg [char(10)]
        room [char(4)]
        manuf [char(10)]
        model [char(10)]
---hpotter---
    customers
        cid [char(4)]
        cname [varchar(13)]
        city [varchar(20)]
        discnt [real(7)]
    agents
        aid [char(3)]
        aname [varchar(13)]
        city [varchar(20)]
        percent [int(10)]
    products
        pid [char(3)]
        pname [varchar(13)]
        city [varchar(20)]
        quantity [int(10)]
        price [real(7)]
    orders
        ordno [int(10)]
        month [char(3)]
        cid [char(4)]
        aid [char(3)]
        pid [char(3)]
        qty [int(10)]
        dollars [real(7)]
...

```

仔细查看以下三段范例代码还是会有好处的，哪怕你并不打算使用其中某种数据库。因为我们会使用不同方法来获取这些信息，所以对你多少都会有所帮助。

通过DBI访问MySQL服务器

以下代码使用了DBI方式来从MySQL服务器获取信息。MySQL的SHOW命令让这个任务变得异常简单：

```

use DBI;

print 'Enter user for connect: ';
chomp(my $user = <STDIN>);
print 'Enter passwd for $user: ';
chomp(my $pw = <STDIN>);

```

```

my $start= 'mysql'; # 先连接到这个数据库

# 连接到开始的 MySQL 数据库
my $dbh = DBI->connect("DBI:mysql:$start",$user,$pw,
                      { RaiseError => 1, ShowErrorStatement => 1 });

# 查询服务器上当前有哪些数据库可以使用
my $sth=$dbh->prepare(q{SHOW DATABASES});
$sth->execute;

my @dbs = ( );
while (my $aref = $sth->fetchrow_arrayref) {
    push(@dbs,$aref->[0]);
}

# 查询每个数据库内有哪些表可以使用
foreach my $db (@dbs) {
    print "---$db---\n";

    $sth=$dbh->prepare(qq{SHOW TABLES FROM $db});
    $sth->execute;

    my @tables=( );
    while (my $aref = $sth->fetchrow_arrayref) {
        push(@tables,$aref->[0]);
    }

    # 查询每张表都有哪些字段
    foreach my $table (@tables) {
        print "\t$table\n";

        $sth=$dbh->prepare(qq{SHOW COLUMNS FROM $table FROM $db});
        $sth->execute;

        while (my $aref = $sth->fetchrow_arrayref) {
            print "\t\t", $aref->[0], ' [' , $aref->[1], "]\n";
        }
    }
}
$dbh->disconnect;

```

关于这段代码的简明注解：

- MySQL 5.x（在写本书时算比较新的版本）引入了一个比较特殊的元数据数据库，名为INFORMATION_SCHEMA，其中含有很多能用标准SELECT语句查询的特殊表。使用它们能获得和使用SHOW命令等同的信息。如果你在使用5.x版本的MySQL，那么也可以使用这个新办法来获取表和字段信息。不过查询这个数据库比查询其他数据库要慢些，所以如果你非常关注性能的话请先三思。
- 在这里连接到开始的数据库只是为了满足DBI连接语法的需要，其实SHOW命令并不需要预先登录数据库。

- 如果你觉得SHOW TABLES和SHOW COLUMNS命令中的变量可以用占位符来代替的话，你确实很专业。不过这里MySQL的DBD程序/服务器组合驱动并不支持占位符（起码在本书编写的时候还不支持）。如果你可以在类似的场合使用占位符，请务必尝试一下。因为在这里占位符能够防止SQL注入攻击的发生，主要是因为（前面提到过的）自动引用机制。
- 我们这里使用命令行提示用户输入用户名和密码，这是为了避免代码（中硬编码的信息）泄露机密，也避免了进程调用机制可能会（通过进程查看机制）泄露的机密。命令行提示会把密码显示在输入终端，可以考虑使用Term::Readkey来获得更好的安全显示机制。
- 另外我们还有一个来自Tim Bunce本人的技巧。注意我们在连接数据库时设置了RaiseError和ShowErrorStatement参数。这会让DBI自动检查并汇报错误，省去了我们在每次DBI调用之后的or die “something” 判断。这能让代码简洁很多。

通过DBI访问Oracle服务器

下面列出的是Oracle版本的代码。这个例子后面有很多注解，所以请仔细对照代码阅读：

```
use DBI;
use DBD::Oracle qw(:ora_session_modes);

print 'Enter passwd for sys: ';
chomp(my $pw = <STDIN>);

my $dbh =
    DBI->connect( 'DBI:Oracle:perlsysadm', 'sys', $pw,
        { RaiseError => 1, AutoCommit => 0, ora_session_mode => ORA_SYSDBA } );

my ( $catalog, $schema, $name, $type, $remarks ); # table_info 会返回这些信息
my $sth = $dbh->table_info( undef, undef, undef, 'TABLE' );

my (@tables);

while ( ( $catalog, $schema, $name, $type, $remarks ) = $sth->fetchrow_array() )
{
    push( @tables, [ $schema, $name ] );
}

for my $table ( sort @tables ) {
    $sth = $dbh->column_info( undef, $table->[0], $table->[1], undef );

    # 如果在下面的fetchrow_arrayref()碰到ORA-24345错误，
    # 则可以像DBD::Oracle文档中说明的那样，在此处设置 $sth->{LongTruncOk} = 1

    print join( '.', @$table ), "\n";
    while ( my $aref = $sth->fetchrow_arrayref ) {
```

```

        # [3] = COLUMN_NAME, [5] = TYPE_NAME, [6] = COLUMN_SIZE
        print "\t\t", $aref->[3], ' ', lc $aref->[5], "(", $aref->[6], ")\n";
    }
}

$sth->finish;
$dbh->disconnect;

```

下面就是之前承诺给出的注解：

- 首先是幕后信息：Oracle对“数据库”一词的理解和其他数据库不同。大多数其他数据库都允许某个用户拥有一个数据库，此用户在数据库中有权限创建表。这是为什么前面的例子先查找服务器上所有的数据库，然后遍历每个数据库，从而列出其中的表。而Oracle并没有类似的层次概念。没错，Oracle也有数据库的说法，但它指的更多的是存储单元，往往与具体用户或者用户的表无关。最接近其他数据库的“数据库”概念的应该是Oracle的“模式(Schema)”。模式是某个用户所拥有对象的集合（包括表、索引等等）。表通常用SCHEMA.TABLENAME的方式制定。上面的代码连接到一个名为“perlsysadm”的单一数据库实例并展示其中的内容。
- 理想情况下这段代码应该使用具有适当权限的账户来连接。但为了让代码更加简单、更加通用，我们尝试使用Oracle的sys用户来登录。这个用户有查看数据库所有表的权限。为了用这个用户的身份登录，连接的时候必须使用特殊的SYSDBA权限，而ora_session_mode => ORA_SYSDBA 这个有趣的参数就是为此设立的。如果你正好有一个专门为此任务创建的账户，那么请修改代码，避免使用“全知全能”的sys账户。
- 除数据库连接代码之外，这段代码基本和服务器类型无关。不像之前的MySQL代码使用了SHOW命令来完成任务，这里我们使用了标准的DBI table_info()和column_info()调用来获取需要的信息。Oracle自己有类似的命令(DESCR tablename)，可以用来查询表结构，但使用更通用的方法，能更好地做到跨平台移植。
- 这里的范例代码其实做了些画蛇添足的事情。为了保持之前范例中的代码结构，这段代码先查出所有表，然后逐个查出表中所有的字段。但其实column_info()方法能够直接查出数据库中所有表的所有字段信息，只要调用时省略方案名和表名即可（也就是说这样调用：column_info(undef,undef,undef,undef)）。另外，DBI规范还说结果信息的返回是经过排序的，所以sort()调用也可以省略。

通过ODBC访问Microsoft SQL Server

基于DBI/DBD::ODBC的代码基本上综合了之前两种方式来获取Microsoft SQL Server数据

库的信息。首先我们使用特定数据库查询^[注5]来获取数据库列表，然后我们使用标准DBI调用`table_info()`和`column_info()`来获取需要的其他信息。

这里一个显著的改变是初始连接字符串：这里使用`'dbi:ODBC:{DSN_name_here}'`作为`connect()`的参数，同时使用了另一个有特权的用户（请看下面的“注意”），还移除了`ora_session_mode`参数。

注意：在SQL Server 2000和SQL Server 2005之间有一个显著的差异，那就是元数据（即所有对象的列表等）的可见性。就2000版本而言，系统中的任何用户都可以查看这个信息，但是到了2005版本之后，只有获得了VIEW ANY DEFINITION权限的用户才能查看同样的信息。

这些差异改变了程序：

```
use DBI;

# 这里的代码假设能访问数据库的用户叫做 mssqldb,
# 而你实际可用的数据库用户名应该不同于此
print 'Enter passwd for mssqldb: ';
chomp(my $pw = <STDIN>);

# 假设之前已经定义好叫做 “PerlSys” 的 DSN
my $dbh =
    DBI->connect( 'dbi:ODBC:PerlSys', 'mssqldb', $pw, { RaiseError => 1 });

# 提取所有数据库的名称
my (@dbs) =
    map { $_->[0] }
    @{ $dbh->selectall_arrayref("select name from master.dbo.sysdatabases") };

my ( $catalog, $schema, $name, $type, $remarks ); # table_info 会返回这些信息
foreach my $db (@dbs) {

    my $sth = $dbh->table_info( $db, undef, undef, 'TABLE' );

    my (@tables);

    while ( ( $catalog, $schema, $name, $type, $remarks ) =
        $sth->fetchrow_array() ) {
        push( @tables, [ $schema, $name ] );
    }

    for my $table ( sort @tables ) {
        $sth = $dbh->column_info( $db, $table->[0], $table->[1], undef );
        print join( '.', @$table ), "\n";
        while ( my $aref = $sth->fetchrow_arrayref ) {
```

注5：如果用户有权限访问服务器上的所有数据库，而你也不希望和系统表打交道，那么可以使用 `select catalog_name from information_schema.schemata` 这条查询语句来获取同样的信息，不过这需要版本比较新的 SQL Server。

```

# [3] = COLUMN_NAME, [5] = TYPE_NAME, [6] = COLUMN_SIZE
print "\t\t", $aref->[3], ' ', lc $aref->[5], "(", $aref->[6],
    ")]\n";
    }
}
}
$dbh->disconnect;

```

下面的代码是为了扩展思路而写的，它使用了比较老的Win32::ODBC模块。这段代码和之前的例子有些差异。首先，它使用了ODBC风格的信息获取方式（请参考Win32::ODBC模块说明文档）。另外还有些不同的地方，比如对特殊的服务器存储过程的调用（比如sp_columns()），而且调用语法也很古怪。这个例子是为了展示Win32::ODBC的使用方式而加入的，也许什么时候你会用上它。

下面就是代码：

```

use Win32::ODBC;

print 'Enter user for connect: ';
chomp(my $user = <STDIN>);
print 'Enter passwd for $user: ';
chomp(my $pw = <STDIN>);

my $dsn='sysadm'; # 将要使用的 DSN 的名称

# 查找可用的 DSN，如果没有就创建一个新的 $dsn
die 'Unable to query available DSN's'.Win32::ODBC::Error()."\n"
    unless (my %dsnavail = Win32::ODBC::DataSources());
if (!defined $dsnavail{$dsn}) {
    die 'unable to create DSN:'.Win32::ODBC::Error()."\n"
        unless (Win32::ODBC::ConfigDSN(ODBC_ADD_DSN,
            "SQL Server",
            ("DSN=$dsn",
            "DESCRIPTION=DSN for PerlSysAdm",
            "SERVER=mssql.happy.edu",
            "DATABASE=master",
            "NETWORK=DBMSSOCN", # TCP/IP Socket Lib
            )));
}

# 通过刚才定义的DSN连接到master数据库
#
# 由于在DSN中已经指定了DATABASE=master，所以接下来不用
# 显式选择起始连接的数据库
my $dbh = new Win32::ODBC("DSN=$dsn;UID=$user;PWD=$pw;");
die "Unable to connect to DSN $dsn:".Win32::ODBC::Error()."\n"
    unless (defined $dbh);

# 查找服务器上已经建好的数据库，如果失败Sql会返回错误代号
if (defined $dbh->Sql(q{SELECT name from sysdatabases})){
    die 'Unable to query databases:'.Win32::ODBC::Error()."\n";
}

```

```

my @dbs = ( );
my @tables = ( );
my @cols = ( );
# ODBC 需要分两步走, 先提取数据,
# 然后通过特殊的方法调用 (Data) 来访问这些数据
while ($dbh->FetchRow()){
    push(@dbs, $dbh->Data("name"));
}
$dbh->DropCursor(); # 这就好比是 DBI 的 $sth->finish()

# 查找每个数据库的 user 表
foreach my $db (@dbs) {
    if (defined $dbh->Sql("use $db")){
        die "Unable to change to database $db:" .
            Win32::ODBC::Error() . "\n";
    }
    print "---$db---\n";
    @tables=();
    if (defined $dbh->Sql(q{SELECT name from sysobjects
        WHERE type="U"})){
        die "Unable to query tables in $db:" .
            Win32::ODBC::Error() . "\n";
    }
    while ($dbh->FetchRow()) {
        push(@tables,$dbh->Data("name"));
    }
    $dbh->DropCursor();

    # 查找每张表的字段信息
    foreach $table (@tables) {
        print "\t$table\n";
        if (defined $dbh->Sql(" {call sp_columns (\'$table\')} ")){
            die "Unable to query columns in $table:" .
                Win32::ODBC::Error() . "\n";
        }
        while ($dbh->FetchRow()) {
            @cols=$dbh->Data("COLUMN_NAME","TYPE_NAME","PRECISION");
            print "\t\t", $cols[0], " [", $cols[1], "(", $cols[2], ")]\n";
        }
        $dbh->DropCursor();
    }
}
$dbh->Close();

die "Unable to delete DSN:".Win32::ODBC::Error()."\n"
unless (Win32::ODBC::ConfigDSN(ODBC_REMOVE_DSN,
    "SQL Server", "DSN=$dsn"));

```

登录数据库

前面介绍过, 数据库管理员需要面对和系统管理员类似的问题, 如维护用户的登录与账户。举例来说, 我作为一名教数据库开发的讲师, 常常需要为新学员开设我的Oracle

数据库的账户，并且要给他们设置合理的（学习用的）数据库空间限额。以下简化过的代码能帮助我们创建数据库账户：

```
use DBI;

my $userquota    = 10000; # 每个用户给定的用户空间限额，单位为 K
my $usertmpquota = 2000; # 每个用户给定的临时数据库表空间限额，单位为 K

my $admin = 'system';
print "Enter passwd for $admin: ";
chomp(my $pw = <STDIN>);
my $user=$ARGV[0];

# 通过将用户名倒序假造一个初始密码，
# 如果用户名不足6个字符，则在密码末尾用减号补足
# 注意：这样的构造算法实在很糟糕，最好改用类似
# Crypt::GeneratePassword 这样的模块自动生成
my $genpass = reverse($user) . '-' x (6-length($user));

my $dbh = DBI->connect("dbi:Oracle:instance",$admin,$pw,{PrintError => 0});
die "Unable to connect: $DBI::errstr\n"
    unless (defined $dbh);

# 准备测试该用户名是否已经存在
my $sth = $dbh->prepare(q{SELECT USERNAME FROM dba_users WHERE USERNAME = ?})
    or die 'Unable to prepare user test SQL: '.$dbh->errstr."\n";

my $res = $sth->execute(uc $user);
$sth->fetchrow_array;
die "user $user exists, quitting" if ($sth->rows > 0);
if (!defined $dbh->do (
    qq {
        CREATE USER ${LOGIN} PROFILE DEFAULT
        IDENTIFIED BY ${PASSWORD}
        DEFAULT TABLESPACE USERS TEMPORARY TABLESPACE TEMP
        QUOTA $usertmpquota K ON TEMP QUOTA $userquota K ON USERS
        ACCOUNT UNLOCK
    })){
    die 'Unable to create database:'.$dbh->errstr."\n";
}

# 分配必要的用户权限
$dbh->do("GRANT CONNECT TO ${LOGIN}") or
    die "Unable to grant connect privs to ${LOGIN}:".$dbh->errstr."\n";

# 可能更好一点的实现方式是显式分配用户所必需的 RESOURCE 权限，
# 而不是一下子把所有权限都分配给他，如删除数据的
# UNLIMITED TABLESPACE
$dbh->do("GRANT RESOURCE TO ${LOGIN}") or
    die "Unable to grant resource privs to ${LOGIN}:".$dbh->errstr."\n";

# 设置恰当的角色
$dbh->do("ALTER USER ${LOGIN} DEFAULT ROLE ALL") or
    die "Unable to use set correct roles for ${LOGIN}:".$dbh->errstr."\n";
```

```
# 确认限额已经生效
$dbh->do("REVOKE UNLIMITED TABLESPACE FROM ${LOGIN}") or
    die "Unable to revoke unlimited tablespace from ${LOGIN}:".$dbh->errstr."\n";

$dbh->disconnect;
```

在课程结束以后，我们可以使用类似的脚本删除账户以及它们的数据库：

```
use DBI;

$admin = 'system';
print "Enter passwd for $admin: ";
chomp(my $pw = <STDIN>);
my $user=$ARGV[0];

my $dbh = DBI->connect("dbi:Oracle:instance",$admin,$pw,{PrintError => 0});
die "Unable to connect: $DBI::errstr\n"
    if (!defined $dbh);

die "Unable to drop user ${user}:".$dbh->errstr."\n"
    if (!defined $dbh->do("DROP USER ${user} CASCADE"));

$dbh->disconnect;
```

你可能会发现其他与账户相关的管理需求。下面是一些可能的想法：

密码检查器

连接至一台服务器并获取数据库和账户列表。尝试使用脆弱的密码（账户名、空密码、默认密码）来连接。

账户映射

生成一个指出哪些账户能访问哪些数据库的列表。

密码控制

开发一个虚构密码(pseudo-password)的期限管理系统。

监控数据库服务器上的空间使用

作为最后一个例子，我们会尝试监控一个SQL服务器上的空间使用情况。这种类型的监控程序有些类似于会在第13章介绍的网络服务监控程序。

从技术上来讲，数据库服务器就是为了存放资料而设立的。这样说，发现服务器上没空余空间存放资料应该是一件糟糕的事情。所以那些能监控空间分配情况的程序就显得非常有用。让我们看看在Oracle数据库上用来监控空间的DBI程序。

下面是程序的输出，它使用图形化的方式展示了用户已用的空间和预定义的限额之间的

关系。每个用户在USERS和TEMP表空间中被分配的限额使用百分比都会用条形图的形式展现出来。下面图表中的u代表用户表空间，而t代表临时表空间。另外对于每个条形，我们都显示了具体的已用空间百分比和总可用空间：

下面的代码能产生这个输出：

```

while ( defined $sth->fetch ) {
    $qdata{$user}->{$tablename} = [ $bytes_used, $bytes_quota ];
}

$dbh->disconnect;

# 将这些信息按照报表图形显示
foreach my $user ( sort keys %qdata ) {
    graph(
        $user,
        $qdata{$user}->{'USERS'}[0],    # 已用字节
        $qdata{$user}->{'TEMP'}[0],
        $qdata{$user}->{'USERS'}[1],    # 限额大小
        $qdata{$user}->{'TEMP'}[1]
    );
}

# 打印图表, 给出对应的用户名、用户与临时空间大小以及使用情况
sub graph {
    my ( $user, $user_used, $temp_used, $user_quota, $temp_quota ) = @_;

    # 表示用户空间使用状况的线
    if ( $user_quota > 0 ) {
        print ' ' x 15 . '|'
            . 'd' x POSIX::ceil( 49 * ( $user_used / $user_quota ) )
            . ' ' x ( 49 - POSIX::ceil( 49 * ( $user_used / $user_quota ) ) )
            . '|';

        # 已使用空间占限额的百分比
        printf( "%.2f", ( $user_used / $user_quota * 100 ) );
        print "%/" . ( $user_quota / 1024 / 1000 ) . "MB\n";
    }

    # 某些用户可能没有设定限额
    else {
        print ' ' x 15 . '|- no user quota' . ( ' ' x 34 ) . "|\n";
    }

    print $user . '-' x ( 14 - length($user) ) . '-|' . ( ' ' x 49 ) . "|\n";

    # 表示临时空间使用状况的线
    if ( $temp_quota > 0 ) {
        print ' ' x 15 . '|'
            . 't' x POSIX::ceil( 49 * ( $temp_used / $temp_quota ) )
            . ' ' x ( 49 - POSIX::ceil( 49 * ( $temp_used / $temp_quota ) ) )
            . '|';

        # 已使用临时空间占限额的百分比
        printf( "%.2f", ( $temp_used / $temp_quota * 100 ) );
        print "%/" . ( $temp_quota / 1024 / 1000 ) . "MB\n";
    }

    # 某些用户可能没有设定临时空间限额
    else {
        print ' ' x 15 . '|- no temp quota' . ( ' ' x 34 ) . "|\n";
    }
}

```

```

    }
    print "\n";
}

```

编写这样的代码并不难，因为Oracle提供了一个非常好用的视图，名为SYS.DBA_TS_QUOTAS，其中的信息恰恰就是表空间的限额使用信息。这里其实是数据库服务器本身提供了管理的便利，对于其他的服务器来说可就没这么容易了（比如在Sybase上你必须累计所有的段空间才能知道数据库的大小）。

这段小程序只是展示了服务器监控的皮毛而已。其实我们可以多次采集SYS.DBA_TS_QUOTAS的信息并且按时间绘图，这样能更好地动态展示空间使用状况。另外还有很多其他指标可以监控，比如CPU使用状况，又比如很多其他数据库性能指标（cache命中率等等）。市面上不乏“某某数据库调优”那样的书，每本都有很多的可监控指标。就让这些指标成为你的Perl脚本的食物吧。

本章所用模块

模块名	CPAN ID	版本
DBI	TIMB	1.50
DBD:mysql	RUDY	2.9008
DBD:Oracle	PYTHIAN	1.17
DBD:ODBC	JURL	1.13
Win32::ODBC （来自 http://www.roth.net ）	JDB	970208

更多参考资料

网上有许多不错的SQL教程，像<http://www.sqlzoo.net>和<http://www.sqlcourse.com>等等。搜索“SQL tutorial”就会找到一堆这样的文章。大致通读一下便能对数据库查询语言有个大概的了解。

http://home.fnal.gov/~dbox/SQL_API_Portability.html 是一份关于如何与各种流行数据库引擎打交道的扩展指南。虽然它所关注的是编写具有可移植性的代码，就像本章看到的那样，但它依然需要对特定数据库的命令和工作机制有所了解，才能更好地管理服务器。

DBI

<http://dbi.perl.org>是官方的DBI主页。说起来，这上面有好些地方已经过时（Tim Bunce是这么说的），不过这仍然应该成为你造访的第一站。

《Programming the Perl DBI》由Alligator Descartes和Tim Bunce合著（O'Reilly），是一本相当不错的关于DBI的书籍。

<http://gmax.oltrelinux.com/dbirecipes.html>上面列出了许多常见任务中会用到的DBI技巧。

Microsoft SQL Server

关于Microsoft SQL Server的信息，除了官方的软件网站(<http://www.microsoft.com/sql>)之外，在<http://www.sqlserverfaq.com>上也还有许多相关的资源。而来自MS Press的微软给Microsoft SQL Server管理员的培训材料，也是相当不错的参考资料。

ODBC

<http://www.microsoft.com/odbc>包含了微软的ODBC相关的信息。不过你得稍微花点力气才能找到具体问题的答案，因为至少在写本书时，这类信息已经归入他们更大的Data Access and Storage Center相关的体系中去了。不过，想在<http://msdn.microsoft.com>网站上搜索ODBC的话，可以在MDAC SDK相关的ODBC库资料中探寻。

<http://www.roth.net/perl/odbc/>是官方的Win32::ODBC主页。（这个只是传统使用的模块。现在，只要可能，你应该改用DBD::ODBC模块。）

《Win32 Perl Programming: The Standard Extensions》，由Dave Roth编著（Macmillan出版），他是Win32::ODBC的作者，这本书是在Windows中进行Perl编程的优秀参考书籍。

Oracle

Oracle的世界就大多了。有不计其数的关于Oracle的书籍和网站可以参考，我觉得最为有用的是<http://www.orafaq.com>，其中不乏初级的和高级的Oracle解决方案。

位于<http://otn.oracle.com>的文档和教程也是不错的参考资料，对各种Oracle数据库版本都有较为深入的讲解。

以往章节都是讨论如何管理特殊的服务、技术或相关的知识领域。本章略作变化，我们来谈谈在系统管理工作中，怎样用Perl作为工具来处理电子邮件。

电子邮件可以当作一种极佳的消息通知机制：当有问题出现，我们希望能有程序自动通知，报告自动化进程的运行状态和结果（如前夜运行的*cron*任务或其他定期运行的服务），或让我们知道任何我们感兴趣的系统状态变化。本章将要探究如何用Perl发送邮件以完成这样的需求，当然少不了实际动手，看看会有哪些常见错误要小心处理。

我们还要看看如何用Perl来收取邮件并作后续处理，以提供更有价值的信息。在抵御垃圾邮件和管理用户提问邮件方面，Perl也是不可或缺的。

阅读本章之前，我们假设你已经比较全面牢固地掌握了有关邮件的基础知识。此外，还假设你所使用（或能访问）的电子邮件系统遵循IETF规范收发邮件。本章的例子将涉及类似SMTP（Simple Mail Transfer Protocol，简单邮件传输协议，RFC 2821）这样的协议。所使用的邮件格式规范也是和RFC 2822兼容的。这些术语将在本章适当时候予以介绍。

发送邮件

在开始讨论复杂问题之前，我们先来看看发送邮件的基本环节。Unix上最为传统的用Perl发送邮件的代码大致如同下面取自Perl FAQ列表中的范例：

```
# 我们假设当前sendmail命令安装在/usr/sbin目录下
# 不同的操作系统或发布版本上，sendmail的安装目录可能有所不同
open my $SENDMAIL, '|-', '|/usr/sbin/sendmail -oi -t -odq' or
    die "Can't fork for sendmail: $!\n";
print $SENDMAIL <<'EOF';
From: User Originating Mail <me@host>
To: Final Destination <you@otherhost>
Subject: A relevant subject line
```

```
Body of the message goes here after the blank line
in as many lines as you like.

EOF
close(SENDMAIL) or warn "sendmail didn't close nicely";
```

注意：一种常见的错误（早年的Perl4语法允许这么做，所以有些Perl程序员会习惯性犯这种错误）是在双引号引起的字符串中直接写出@符号：

```
$address = "fred@example.com"; # 这实际上会做数组内插
```

这行代码应当改作下面任意一种形式：

```
$address = "fred\@example.com";
$address = 'fred@example.com';
$address = q{ fred@example.com };
$address = join('@', 'fred', 'example.com');
```

像上面调用*sendmail*的代码能够在绝大多数类似的环境中正常工作，可如果在没有安装“sendmail”邮件传输代理的操作系统（比如 Windows）上的话就会报错。如果你用的正是这样的操作系统，不用担心，还是有些办法的。请看接下来几个小节的说明。

获取sendmail（或其他类似的邮件传输代理）

Windows上有各式*sendmail*和类似*sendmail*的工具可以使用，但大多数都是商业程序。如果想要免费的并能当作*sendmail*一样来调用的工具，可以试试Cygwin *exim* (<http://cygwin.com/packages/exim>)。如果想更轻巧些，也愿意稍稍改动下Perl代码的话，可以试试*blat* (<http://www.blatt.net>)，你只需要修改下调用参数就行。

将发送邮件的具体操作交给后端的代理来完成从而简化代码，是这种方案的最大好处。通常邮件传输代理（mail transport agent, MTA）会自动处理邮件发送过程中遇到的各种问题：如果无法连接目标邮件服务器就会重试几次，实在不行就查询邮件交换服务器（Mail eXchange, MX）域名记录，选择恰当的服务器重新发送，必要的话还需要重写邮件头、处理退信等等，诸如此类复杂但不可或缺的工作。所以尽可能放手让MTA来做，不用让Perl操心这些琐事。

使用特定操作系统的IPC框架驱动邮件客户端

在Mac OS X或者Windows系统上，我们可以通过调用系统自身的内部进程通信（interprocess communication, IPC）框架来驱动邮件客户端。

Mac OS X原本就自带安装了Postfix MTA，只是按照最小化需要配置的。如果不想麻烦

地配置，可以用Perl运行AppleScript来使内置的邮件客户端（通常名为*Mail.app*）工作：

```
use MacPerl;

my $to      = 'user@example.com';
my $subject = 'Hi there';
my $body    = 'message body';

MacPerl::DoAppleScript(<<EOAS);
tell application "Mail"
    set theNewMessage to make new outgoing message with properties
        {subject:"$subject", content:"$body", visible:true}
    tell theNewMessage
        make new to recipient at end of to recipients with properties
            {address:"$to"}
        send
    end tell
end tell
EOAS
```

这段代码执行了一段非常简单的AppleScript，功能就是创建和发送邮件。在Mac OS X上有许多办法通过Perl来运行AppleScript，到CPAN搜索“AppleScript”就可以看到。这里所用的模块MacPerl，是Chris Nandor所写的Mac::Carbon模块的一部分，从Mac OS X 10.5以来这个模块就捆绑发行，所以用它会比较方便。

而在Windows中，我们可以用Win32::OLE模块来控制广泛使用的Outlook程序：^[注1]

```
use Win32::OLE;
use Win32::OLE::Const 'Microsoft Outlook';

my $outl = Win32::OLE->new('Outlook.Application');
my $ol   = Win32::OLE::Const->Load($outl);

my $message = $outl->CreateItem(olMailItem);

$message->Recipients->Add('user@example.edu');
$message->{Subject} = 'Perl to Outlook Test';
$message->{Body}    = "Hi there!\n\nLove,\nPerl\n";

$message->Send;
```

为了驱动Outlook工作，我们需要先创建一个Outlook Application对象，并借此创建一封要发送的邮件。同时用Win32::OLE::Const模块导入Outlook相关的OLE常量，所以这里我们可以用olMailItem来创建邮件对象。之后代码的含义就非常显而易见了。

注1： 这里的代码使用了 Application 对象来控制 Outlook。自 Outlook 2000 版本以来，该对象就作为首选的控制方式。本书第一版的发行早于此版本，所以当时使用了低级的 MAPI 调用来实现。而现在的方法则更容易掌握和理解。

上面这段代码非常简单，可要是再增加点功能，如添加附件、移动邮件到其他文件夹等等，就需要到MSDN网站学习如何同Outlook交互，显然这属于多出来的工作。还好，有位叫做Barbie的开发者发布的Mail::Outlook模块可以让我们以下面这种方式运行：

```
use Mail::Outlook;

my $outl = new Mail::Outlook();

my $message = $outl->create();

$message->To('user@example.edu');
$message->Subject('Perl to Outlook Test');
$message->Body("Hi there!\n\nLove,\nPerl\n");
$message->Attach(@files);

$message->send() or die "failed to send message";
```

话说回来，依靠AppleScript或Application对象来实现的方式本质上根本无异于调用“sendmail”程序。具体工作托付给外部程序，多少总存在着依赖性问题，所以这种方式应该是最后考虑使用的。

直接使用邮件协议发送

我们最后的选择，还是自己写代码，按照邮件服务器能够理解的语言同其交互。该语言就是RFC 2821所定义的简单邮件传输协议。下面给出了最基本的SMTP交互过程。所发送的信件内容使用黑体显示：

```
% telnet example.com 25      ——连接到 example.com 邮件服务器的 SMTP 端口
Trying 192.168.1.10 ...
Connected to example.com.
Escape character is '^]'.
220 mailhub.example.com ESMTP Sendmail 8.9.1a/8.9.1; Sun, 13
  Apr 2008 15:32:16 ?0400 (EDT)
HELO client.example.com      ——标识我们从哪台机器发出的连接请求
                               (这里也可以使用 EHLO 命令)
250 mailhub.example.com Hello dnb@client.example.com [192.168.1.11],
  pleased to meet you
MAIL FROM: <dnb@example.com> ——给出发件人地址
250 <dnb@example.com>... Sender ok
RCPT TO: <dnb@example.com>   ——给出收件人地址
250 <dnb@example.com>... Recipient ok
DATA                          ——接下来开始发送邮件内容，请注意，其中包括好几行邮件头
354 Enter mail, end with "." on a line by itself
From: David N. Blank-Edelman <dnb@example.com>
To: dnb@example.com
Subject: SMTP is a fine protocol

Just wanted to drop myself a note to remind myself how much I love SMTP.
```

```
Peace,  
dNb  
.  
250 PAA26624 Message accepted for delivery  
QUIT  
221 mailhub.example.com closing connection  
Connection closed by foreign host.
```

——单独一行小数点标志着邮件内容发送完毕
——结束同服务器的会话，断开连接

要写出像上面这样进行网络通信的脚本并非什么难事。我们可以用`IO::Socket`模块，或是下一章要介绍的`Net::Telnet`模块。但一般情况下，我们不该关心底层交互细节，而应直接使用那些更高级别抽象的电子邮件处理模块，如Jenda Krynicky的`Mail::Sender`模块、Milivoj Ivkovic的`Mail::Sendmail`模块、Graham Barr所写的`MailTools`工具包中的`Mail::Mailer`模块以及`Email::Send`模块（这是Ricardo Signes维护的PEP——Perl Email Project中的模块之一）等等。这四个模块都不依赖特定操作系统，基本上可以到处运行。接下来我们详细讨论`Email::Send`模块的使用。之所以先介绍这个模块，有两方面的原因：一是它为上面所谈到的两种邮件发送方式提供了单一用户接口；二是以此为切入点，一窥PEP中各种邮件模块的全貌。

注意：最新提示：本书出版后，`Email::Send`模块（及其他绝大多数PEP模块）的维护者Ricardo Signes宣称将要废弃`Email::Send`，改用新式的`Email::Sender`模块。但`Email::Sender`尚未全部完成（不过其中附带了一个轻量级的`Email::Sender::Simple`可供使用），而且它所具备的测试也不及`Email::Send`完备。Signes表示他将继续维护`Email::Send`一年，不过我还是建议你关注`Email::Sender`的进展，一旦时机成熟即可改用新式武器。

使用Email::Send发送普通邮件

对于简单的邮件，可以像这样直接把邮件消息存入标量变量，再交给`Email::Send`发送：

```
my $message = <<'EOM';  
From: motherofallthings@example.org  
To: dnb@example.edu  
Subject: advice  
  
I am the mother-of-all-things and all things should wear a sweater.  
  
Love,  
Mom  
EOM
```

直接在程序中写邮件头比较容易出错，有时候会发生不符合RFC规范的情况。我们可以用PEP的`Email::Simple`模块及其内含的`Email::Simple::Creator`插件模块按照面向对象的写法构造邮件消息，它们会帮我们做各项检查，避免出现格式问题。下面先了解该模块的用法，然后再看如何用`Email::Send`发送构造好的邮件。

Email::Simple::Creator所提供的create()方法用起来非常直白，只需两个参数，一个header(包含一个邮件头及其内容的列表)，一个body(一个带有消息正文的标量)，请看：

```
use Email::Simple;
use Email::Simple::Creator;
use Email::Send;

my $message = Email::Simple->create(
    header => [
        From    => 'motherofallthings@example.org',
        To      => 'dnb@example.edu',
        Subject => 'Test Message from Email::Simple::Creator',
    ],
    body => "Hi there!\n\tLove,\n\tNb",
);
```

很简单，不是吗？好，现在来看看如何发送这封邮件。如果要直接通过SMTP发送，可以这么做：

```
my $sender = Email::Send->new({mailer => 'SMTP'});
$sender->mailer_args([Host => 'smtp.example.edu']);
$sender->send($message) or die "Unable to send message!\n";
```

若想让 *sendmail* 代劳，或用任何像 *sendmail* 一样的 MTA 系统（比如 Exim 或 Postfix）代为发送，仅需改成：

```
my $sender = Email::Send->new({mailer => 'Sendmail'});
$Email::Send::Sendmail::SENDMAIL = '/usr/sbin/sendmail';
$sender->send($message) or die "Unable to send message: $!\n";
```

可能你已经注意到了，这里修改了包变量\$Email::Send::Sendmail::SENDMAIL的设置。因为Email::Send::Sendmail不会尝试自动搜索*sendmail*程序的路径，它默认直接使用*sendmail*，只要环境变量的路径设置可以找到就没问题，否则就需要手动指定具体路径。所以为了保险起见，我们还是在这里给出明确的*sendmail*路径。

除了上面两种mailer取值外，还有许多其他可用的发送后端，其名称和Email::Send::模块相对应。我最喜欢的一个是源自Email::Send::Test模块的'test'。对于应用程序来说，Email::Send::Test像是如往常一样能够将邮件发送出去。可实际上它并没投递，而是把这些邮件保存到数组，以方便查验是否符合预期要求，那么用来调试程序时，这可以避免惊扰到实际用户，直到代码测试通过，改回实际可发送邮件的mailer即可。

使用Email::Send发送带附件的邮件

既然用Perl发送邮件如此简单，人们自然就想继续用它来完成更复杂的任务。虽说通过电子邮件传输文件并非最佳方案，但我们真的需要能发送带附件的邮件。具体实施时会

涉及多用途互联网邮件扩展(Multipurpose Internet Mail Extensions, MIME) 标准。它很复杂, 研究它就像掉进兔子洞去冒险一样。为了描述这种扩展标准, 人们制定了RFC 2045、2046、2047、2077、4288和4289 (没错, 至少要用这六篇RFC文档来描绘这头怪物)。简单说来, MIME就是用于描述包含在邮件中各式媒体文件表示方式的标准。

出于篇幅考虑，本章无法展开MIME的各项细节，只能概括介绍。我们只需要认识到符合MIME标准的邮件都由若干独立部分组成，每个部分叫做MIME part，它们有各自的头信息，一般至少包含内容类型、数据编码方式等元数据。我们可以用PEP（由Ricardo Signes维护）的Email::MIME模块来解析处理MIME内容。此外，我们可以用Email::MIME附带的插件模块Email::MIME::Creator（也是由Signes维护）毫不费力地构造出带附件的邮件。先来看代码范例吧，然后再解释它们如何工作：

```

use Email::Simple;
use Email::MIME::Creator;
use File::Slurp qw(slurp);
use Email::Send;

my @mimeparts = (
    Email::MIME->create(
        attributes => {
            content_type => 'text/plain',
            charset      => 'US-ASCII',
        },
        body => "Hi there!\n\tLove,\n\t\t\t\tNb\n",
    ),
    Email::MIME->create(
        attributes => {
            filename      => 'picture.jpg',
            content_type  => 'image/jpeg',
            encoding      => 'base64',
            name          => 'picture.jpg',
        },
        body => scalar slurp('picture.jpg'),
    ),
);

my $message = Email::MIME->create(
    header => [
        From      => 'motherofallthings@example.org',
        To        => 'dnb@example.edu',
        Subject   => 'Test Message from Email::MIME::Creator',
    ],
    parts => [@mimeparts],
);

my $sender = Email::Send->new({mailer => 'Sendmail'});
$Email::Send::Sendmail::SENDMAIL = '/usr/sbin/sendmail';
$sender->send($message) or die "Unable to send message!\n";

```

第一步先构造组成该邮件的两个MIME part: 纯文本的部分用于表示邮件正文, 指定文

件名的部分表示附件。代码依然很直观，只是附件part中指定给body的内容看起来有些古怪。因为Email::MIME->create()方法只能接受用标量表示的内容，所以读取的附件也应该以标量表示。一次性读取某个文件所有内容最便捷高效的办法就是用Dave Rolsky的File::Slurp模块。slurp()默认返回的是包含每一行信息的数组，也就是列表上下文，所以我们这里必须用scalar来告诉它按标量上下文返回完整的文件内容。

接下来就是构造包含这两个MIME part的邮件。还是用Email::MIME->create()创建，给出相关的收件人、标题等头信息，以及刚才准备好的两个MIME part对象。最后，就像之前一样发送该邮件。

使用Email::Send发送HTML格式的邮件

其实我不太愿意演示如何发送HTML邮件，我个人非常讨厌这种臃肿的信息表现方式。不过要是哪天有人要求你这么`做`，就只好另当别论了。所以多少还是需要了解一点发送HTML邮件的编程实现。就算你跟老板反复重申“HTML邮件恶心极了”这样的话，也不会改变他要求你去编码的命运。好吧，为了让你能保住饭碗，这里给你点例子参考，可千万别说是我教给你这门秘笈的哦。

实际上，构造HTML邮件和之前构造纯文本邮件相似，不同的只是正文的编码和类型。所以，我们还是可以用之前的Email::MIME::Creator模块。如果HTML只是些文字或者风格定义的话，编码上基本和纯文本邮件的相同。但如果HTML页面中包含图片或类似媒体数据的话，编码就不那么简单了。把这类数据嵌入邮件一起发送有两方面的好处：一方面，如果用URL的方式链接图片，那么邮件显示起来会比较慢，或者断开网络连接的时候根本就不能显示，这是出于性能考虑；另一方面，许多邮件客户端软件都默认禁止加载基于远程URL的图片资源，以免发送垃圾邮件的人得以确认用户的邮件地址真实有效，这是出于安全考虑。

不必担心，这样的麻烦事已经有BBC的程序员替我们解决了，可以用他们创建并维护的Email::MIME::CreateHTML模块，用法和之前的极为相似。这里是一个简单范例，发送HTML邮件正文的同时附带了可选的纯文本内容：

[illegible]

HTML

```
my $message = Email::MIME->create_html(
    header => [
        From    => 'motherofallthings@example.org',
        To      => 'dnb@example.edu',
        Subject => 'Test Message from Email::MIME::CreateHTML',
    ],
    body => $annoyinghtml,
    text_body => "Hi there!\n\tLove,\n\t\t\t",
);

my $sender = Email::Send->new( { mailer => 'Sendmail' } );
$Email::Send::Sendmail::SENDMAIL = '/usr/sbin/sendmail';
$sender->send($message) or die "Unable to send message!\n";
```

此处代码的实现极为简单，仅是将HTML源代码及对应的纯文本版本的消息存入标量变量中，递交给该模块即可。虽然这里的HTML没有使用外部的图片资源，但如果有的话，Email::MIME->create_html方法会自动解析并提取那些资源，以额外的MIME part的形式封装到邮件对象中来。随后的邮件发送代码也和之前的例子完全一样，因为在设计Email::Send的时候，作者就考虑到了兼容各种邮件对象，所以这里无需任何改动。

继续行文之前，我还想说明一点：Email::MIME::CreateHTML简便易用的背后，也是需要付出一些代价的。它有很多工作需要依赖于其他模块才能完成，而这些模块又各有各的依赖模块，所以安装起来可能会比较费时费力。倒也不是说特别困难，毕竟我们可以用CPAN.pm和CPANPLUS模块，可要是想轻巧些，就得找别的办法来创建HTML邮件了。

发送邮件时的常见错误

学会如何发送邮件后，就可用电子邮件作为一种通知方法了。一旦动手你会很快发现，实际上怎样发送并非重点，有意思的是发送邮件的时机和内容。

本节将通过对照两种完全不同的实现方式来回答上面的问题。随着介绍的深入，我们会发现有时候甚至需要考虑停止发送电子邮件。^[注2]那么，我们开始吧，来看看系统管理程序发送电子邮件时经常会犯的错误。

持续不断发送邮件

迄今为止最常见的问题就是持续不断发送大量邮件。通过脚本发送通知信件固然不错，比如某项服务发生故障时，发送邮件通知或直接转寄到寻呼机通知管理员来处理，都是

注2：当然，我们仍然假设你决定用电子邮件作为最佳的信息沟通方式。不过在动手之前，请记住这种方式可能存在较长延时，同时也不是特别安全，以及其他诸如此类的问题。

不错的办法。但若每5分钟就进行一次程序，每次都不厌其烦发送一次警报邮件，显然不是什么明智之举。那些持续而来的邮件很快就有可能会被当作垃圾邮件自动过滤到别处，结果就是重要的信息也随之淹没。

控制邮件发送的频率

我把这种状况叫做“邮件烽火台”，最容易的解决办法是在程序内部做好时间间隔控制。如果是长时间运行的脚本，可以使用某个变量记录上次发送邮件的时间，以便下次计算间隔：

```
my $last_sent = time;
```

如果程序是通过Unix的*cron*或者Windows的Task Scheduler服务机制启动，每*N*分钟或*N*小时运行一次，可以将当前时间记录到某个单行文件中，在下次运行时读取以计算时间间隔。使用这种方法需要注意做好相关的安全措施，参见第1章的内容。

如何设置时间间隔视具体需求而定。其实不必过分花哨，建议使用指数级变化逐渐放缓，从最开始每分钟一次（ 2^0 ），到每2分钟一次（ 2^1 ），然后每4分钟一次（ 2^2 ），每8分钟一次（ 2^3 ），直至最后达到最大阈值（比如每天一次）。

同样地，如果是两年到期服务通知之类的问题，随着时间逼近需要增加发送的次数，也可以使用指数变化逐渐加快频率。比如从开始的最大延时每天一次，慢慢逼近到最小延时每5分钟一次。

控制邮件发送的数量

除了重复发送邮件外，如果有许多报警程序一起发送通知邮件，也会让系统管理员发狂。假如网络中所有服务器都在固定时刻发送通知邮件给你，你就有可能错过其中重要的一封。较为稳妥的做法是让它们都往一个中心信息存储点发送邮件通知，^[注3]稍后再将这些信息整合在一封邮件中发送。

让我们来看一个假想的例子。网络中有若干台服务器，且都能往某个共享目录中写入一

注3： Unix 系统中有个很棒的集中化记录程序工作状态的工具，叫做 *syslog*（此外还有一些从它派生而来的类似工具，如 *syslog-ng* 等）。不过，想要高效利用它还得做许多配置工作。不管是技术上的还是管理上的原因，这都不是唯一的选择，所以本章将要展示另外的实现方法。至于如何处理 *syslog* 所记录的日志信息，可以参考第10章。

个单行文本文件来记录工作状态。^[注4]文件名使用服务器名，所包含的内容是前一天晚上科学计算结果的概述。假设这一行以下面的形式记录：

```
hostname success-or-failure number-of-computations-completed
```

下面是收集这些文件信息并整合到一起发送邮件的程序：

```
use Email::Simple;
use Email::Simple::Creator;
use Email::Send;
use Text::Wrap;
use File::Spec;

# 应当提供报告的服务器清单文件
my $repolist = '/project/machinelist';

# 保存状态文件的共享目录
my $reporidir = '/project/reportddir';

# 发送邮件时 From 头所使用的邮件地址
my $reportfromaddr = 'project@example.com';

# 收件人地址
my $reporttoaddr = 'project@example.com';

my $statfile;      # 记录状态报告的文件的名称
my $report;        # 每个状态文件中的报告行
my %success;       # 计算成功的主机
my %fail;          # 计算失败的主机
my %missing;       # 未提交报告的主机清单

# 现在读取服务器清单文件，存入哈希。
# 之后，从该哈希中清除找到状态报告的服务器，
# 那么最后剩下的就是没有提供报告的服务器。

open my $LIST, '<', $repolist or die "Unable to open list $repolist:!\n";
while (<$LIST>) {
    chomp;
    $missing{$_} = 1;
}
close $LIST;

# 应当报告状态的服务器总数
my $machines = scalar keys %missing;

# 从共享目录中读取所有状态文件。
# 注意：此目录中的文件应当由另外的脚本自动定时清空。
opendir my $REPO, $reporidir or die "Unable to open dir $reporidir:!\n";

while ( defined( $statfile = readdir($REPO) ) ) {
```

注4：除此之外我们还可以用数据库记录状态信息。或者只是简单地把这些邮件都发送到某个特定邮箱，然后用另外的 Perl 程序通过 POP3 获取邮件并做后续处理。

```

next unless -f File::Spec->catfile( $repodir, $statfile );

# 打开每个状态文件, 读取单行状态报告, 按既定格式分别取出三段信息
open my $STAT, File::Spec->catfile( $repodir, $statfile )
    or die "Unable to open $statfile: $!\n";

chomp( $report = <$STAT> );

my ( $hostname, $result, $details ) = split( ' ', $report, 3 );

warn "$statfile said it was generated by $hostname!\n"
    if ( $hostname ne $statfile );

# 从未提供报告服务器清单中划去该主机
delete $missing{$hostname};

# 根据成功或者失败状态, 将该主机分别存入不同的哈希
if ( $result eq 'success' ) {
    $success{$hostname} = $details;
}
else {
    $fail{$hostname} = $details;
}
close $STAT;
# 处理完成后, 我们即可在此处删除 $statfile, 为明晚报告做好准备。
# 如果本程序发生故障, 报告就无法再次生成, 所以建议另外运行清空脚本
}
closedir $REPO;

# 构造邮件主题, 提供概要信息
my $subject;
if ( scalar keys %success == $machines ) {
    $subject = "[report] Success: $machines";
}
elsif ( scalar keys %fail == $machines or
        scalar keys %missing >= $machines ) {
    $subject = "[report] Fail: $machines";
}
else {
    $subject
        = '[report] Partial: '
        . keys(%success)
        . ' ACK, '
        . keys(%fail) . ' NACK'
        . ( (%missing) ? ', ' . keys(%missing) . ' MIA' : '' );
}

# 创建邮件正文
my $body = "Run report from $0 on " . scalar localtime(time) . "\n";

if ( keys %success ) {
    $body .= "\n==Succeeded==\n";
    foreach my $hostname ( sort keys %success ) {
        $body .= "$hostname: $success{$hostname}\n";
    }
}

```

```

}

if ( keys %fail ) {
    $body .= "\n==Failed==\n";
    foreach my $hostname ( sort keys %fail ) {
        $body .= "$hostname: $fail{$hostname}\n";
    }
}

if ( keys %missing ) {
    $body .= "\n==Missing==\n";
    $body .= wrap( ' ', ' ', join( ' ', sort keys %missing ) ), "\n";
}

my $message = Email::Simple->create(
    header => [
        From    => $reportfromaddr,
        To      => $reporttoaddr,
        Subject => $subject,
    ],
    body => $body,
);

my $sender = Email::Send->new( { mailer => 'Sendmail' } );
$Email::Send::Sendmail::SENDMAIL = '/usr/sbin/sendmail';
$sender->send($message) or die "Unable to send message!\n";

```

该程序先是读取应提供报告的服务器清单并存入哈希，再依次检查每台服务器是否都记录了状态文件，然后逐个打开状态文件，提取其中分段信息，最后构造一封邮件统一报告所有服务器的运行情况。

下面是最终发出的邮件的样子：

```

Date: Mon, 14 Apr 2008 13:06:09 ?0400 (EDT)
Message-Id: <200804141706.NAA08780@example.com>
Subject: [report] Partial: 3 ACK, 4 NACK, 1 MIA
To: project@example.com
From: project@example.com

Run report from reportscript on Mon Apr 14 13:06:08 2008

==Succeeded==
barney: computed 23123 oogatrons
betty: computed 6745634 oogatrons
fréd: computed 56344 oogatrons

==Failed==
bambam: computed 0 oogatrons
dino: computed 0 oogatrons
pebbles: computed 0 oogatrons
wilma: computed 0 oogatrons

==Missing==

```

mrslate

像上面这样收集信息的方式，也可以通过创建自己的日志记录守护进程，让所有服务器通过网络socket发送状态报告来实现。下面先来看看守护进程服务的代码，本例沿用了之前部分代码，但仍请逐行阅读，之后我们将对新增的重要部分加以讨论：

```
use IO::Socket;
use Text::Wrap;    # 借此模块让打印出来的文本格式更漂亮

# 应当提供报告的服务器清单文件
my $repolist = '/project/machinelist';

# 客户端应该连接的通信端口
my $serverport = '9967';

my %success;      # 计算成功的主机
my %fail;         # 计算失败的主机
my %missing;      # 未提交报告的主机清单

# 载入服务器清单，借助哈希片段的方式对所有主机赋予初值 undef，
# 最后得到形同这样的赋值结果：
# %missing = { key1 => undef, key2 => undef, ...}
@missing{ loadmachines() } = ();
my $machines = keys %missing;

# 设置服务器端 socket 参数
my $reserver = IO::Socket::INET->new(
    LocalPort => $serverport,
    Proto     => "tcp",
    Type      => SOCK_STREAM,
    Listen    => 5,
    Reuse     => 1
) or die "Unable to build our socket half: $!\n";

# 开始监听端口，一旦有连接则继续下面的代码
while ( my ( $connectsock, $connectaddr ) = $reserver->accept() ) {

    # 连接过来的客户端名称
    my $connectname
        = gethostbyaddr( ( sockaddr_in($connectaddr) )[1], AF_INET );

    chomp( my $report = $connectsock->getline );

    my ( $hostname, $result, $details ) = split( ' ', $report, 3 );

    # 如果要求输出汇总报告，则回应相应的邮件主题和正文，
    # 并重新初始化所有的哈希和计数器
    if ( $hostname eq 'DUMPNOW' ) {
        printmail($connectsock);
        close $connectsock;
        undef %success;
        undef %fail;
        undef %missing;
    }
}
```

```

        @missing{ loadmachines() } = ();      # 重新加载服务器清单
        my $machines = keys %missing;
        next;
    }

    warn "$connectname said it was generated by $hostname!\n"
        if ( $hostname ne $connectname );

    delete $missing{$hostname};

    if ( $result eq 'success' ) {
        $success{$hostname} = $details;
    }
    else {
        $fail{$hostname} = $details;
    }
    close $connectsock;
}
close $reserver;

# 打印准备好发送的邮件消息,
# 第一行是邮件主题, 后续所有行是邮件正文
sub printmail {
    my $socket = shift;

    my $subject;
    if ( keys %success == $machines ) {
        $subject = "[report] Success: $machines";
    }
    elsif ( keys %fail == $machines or keys %missing >= $machines ) {
        $subject = "[report] Fail: $machines";
    }
    else {
        $subject
            = '[report] Partial: '
            . keys(%success)
            . ' ACK, '
            . keys(%fail) . " NACK"
            . ( (%missing) ? ', ' . keys(%missing) . ' MIA' : '' );
    }

    print $socket "$subject\n";

    print $socket "Run report from $0 on " . scalar localtime(time) . "\n";

    if ( keys %success ) {
        print $socket "\n==Succeeded==\n";
        foreach my $hostname ( sort keys %success ) {
            print $socket "$hostname: $success{$hostname}\n";
        }
    }

    if ( keys %fail ) {
        print $socket "\n==Failed==\n";
        foreach my $hostname ( sort keys %fail ) {

```

```

        print $socket "$hostname: $fail{$hostname}\n";
    }
}

if ( keys %missing ) {
    print $socket "\n==Missing==\n";
    print $socket wrap( ' ', ' ', join( ' ', sort keys %missing ) ), "\n";
}

}

# 从指定文件加载服务器清单
sub loadmachines {
    my @missing;
    open my $LIST, '<', $repolist or die "Unable to open list $repolist:!\n";
    while (<$LIST>) {
        chomp;
        push( @missing, $_ );
    }
    close $LIST;
    return @missing;
}

```

除了将部分代码移入相应子例程外，主要的变化是网络通信部分。IO::Socket模块负责socket通信，简单易用。socket的概念有点像打电话，我们先是在服务器端设置好socket（IO::Socket->new()），好比电话开机，然后等待来自网络中客户端的连接请求（IO::Socket->accept()）。此时我们的程序将进入暂停状态（或者说待机状态），直到有连接进来，accept()方法返回相应信息。之后我们提取并保存客户端名字，再从socket按行读取输入信息。

所读入的这行信息应该像之前约定的格式分成三段，这里唯一取巧的地方是使用伪主机名DUMPNOW作为命令，让服务器端返回汇总报告邮件并重置哈希和计数器。发送此命令的客户端继而负责将此邮件发送出去。接下来看看客户端代码的实现：

```

use IO::Socket;

# 应当连接的通信端口
my $serverport = '9967';

# 中心服务器的名称
my $servername = 'reportserver';

# 查找网络中对应该服务器名的实际 IP 地址
my $serveraddr = inet_ntoa( scalar gethostbyname($servername) );
my $reportfromaddr = 'project@example.com';
my $reporttoaddr = 'project@example.com';

my $reserver = IO::Socket::INET->new(
    PeerAddr => $serveraddr,
    PeerPort => $serverport,
    Proto    => 'tcp',

```

```

    Type      => SOCK_STREAM
) or die "Unable to build our socket half: $!\n";

if ( $ARGV[0] ne '-m' ) {
    print $reserver $ARGV[0];
}
else {

    # 下面通过use语句加载的模块实际上在开始运行本程序时就会完成加载，
    # 就算还没运行到这里也是如此。如果要简单引入外部代码，可以使用
    # require/import (就像我们在第3章介绍的那样)，不过此处这么写的目的
    # 是想说明，仅当运行程序时给出 -m 开关，才会用到这三个模块。

    use Email::Simple;
    use Email::Simple::Creator;
    use Email::Send;

    print $reserver "DUMPNOW\n";
    chomp( my $subject = <$reserver> );
    my $body = join( ' ', <$reserver> );

    my $message = Email::Simple->create(
        header => [
            From      => $reportfromaddr,
            To        => $reporttoaddr,
            Subject   => $subject,
        ],
        body => $body,
    );

    my $sender = Email::Send->new( { mailer => 'Sendmail' } );
    $Email::Send::Sendmail::SENDMAIL = '/usr/sbin/sendmail';
    $sender->send($message) or die "Unable to send message!\n";
}

close $reserver;

```

首先，打开连接到服务器的socket，大部分时候我们直接将本地运算结果状态传递过去（即来自命令行的参数\$ARGV[0]，比如script.pl "dino fail computed 0 oogatrons"）并关闭连接。不过真要是用这种客户端/服务器端方式记录日志，最好把客户端代码封装到子例程中，然后在预处理完成后调用该子例程。

如果传递给该脚本的参数是-m标志，则发送“DUMPNOW”命令到服务器，然后读取汇总报告邮件的主题和正文，再交给Email::Send发送，具体发送代码我们在之前已见过范例。

为了讨论方便，以上代码都尽可能短小，没有错误处理，没有输入检查，没有访问控制，没有用户认证（任何在网络中的人都可以发送信息给我们的服务器，或者获取报告），也没有持久存储数据（万一服务器失灵了该怎么办？），或者任何其他该注意和处理的意外状况。而且在上面的代码中，服务器只能在同一时刻处理一个请

求，若是数据传输过程中客户端挂起，服务器也就完了。更为周全的服务器代码范例，推荐阅读Lincoln Stein的《Network Programming With Perl》（Addison-Wesley出版）、Tom Christiansen和Nathan Torkington的《Perl Cookbook》(<http://oreilly.com/catalog/9780596003135/>)（O'Reilly出版）中有关client/server的论述。Jochen Wiedmann的Net::Daemon模块也可以帮你构建稳健的守护进程程序。

关于管理和控制邮件发送数量的思路大概就是这些，接下来我们继续看看其他常见问题。

无用主题行

邮件主题Subject行如果不加以善用，简直是暴殄天物。特别是自动发送的邮件更是如此，最好每封邮件都生成一个包含一定信息量的Subject。如果收件箱里全是如下这样的邮件你会怎么想：

```
Super-User      File history database merge report
Super-User      File history database merge report
Super-User      File history database merge report
Super-User      File history database merge report
Super-User      File history database merge report
Super-User      File history database merge report
Super-User      File history database merge report
```

如果变成这样，会不会好些：

```
Super-User      Backup OK, 1 tape, 1.400 GB written.
Super-User      Backup OK, 1 tape, 1.768 GB written.
Super-User      Backup OK, 1 tape, 2.294 GB written.
Super-User      Backup OK, 1 tape, 2.817 GB written.
Super-User      Backup OK, 1 tape, 3.438 GB written.
Super-User      Backup OK, 3 tapes, 75.40 GB written.
```

抑或如此，是不是更好呢：

```
Super-User      Backup of Hostname OK, 1 tape, 1.400 GB written.
Super-User      Backup of Hostname:/usr OK, 1 tape, 1.768 GB written.
```

Subject行应当提供精准客观的概述。从中我们应当让我们可以非常清楚地获知状态报告中各种成败或其他状况的概要。花一点儿时间改进主题可是能省下许多仔细查阅邮件的时间哦。

消息正文中信息不足

较之Subject行，邮件正文可以更为详尽些。如果要报告的是发生的故障或错误，则应力求提供详细的可供诊断和处理的信息。归结起来应该需要回答以下几个问题：

谁发送的报告？

生成报告脚本是哪一个？可以在代码中使用`$0`（如果没有写明路径）显示当前脚本的完整路径。如果有版本号也请说明。

故障点在何处？

在脚本中给出错误发生的地方。Perl内置函数`caller()`能够返回各种有用信息：

```
# 注意：不同Perl版本中caller()所返回的内容可能有所不同，
# 所以请查阅perlfunc文档确定实际用法
(package, $filename, $line, $subroutine, $hasargs, $wantarray,
 $evaltext, $is_require) = caller($frames);
```

`$frames`是堆栈帧的数量（比如在子例程中调用子例程）。通常我们只用到`$frames`为1，即获取上级子例程调用所在点的信息。下面给出的是之前服务器代码范例中间部分调用`caller()`时返回的内容：

```
('main', 'repserver', 32, 'main::printmail', 1, undef)
```

它表示当前脚本在运行到第32的文件名`repserver`时，仍然从属于`main`包。在该点所执行的是`main::printmail`子例程（调用该子例程时给出了参数，运行结束应该按照标量上下文返回数据）。

若是对阅读邮件报告的人更体贴些，可以结合`caller()`和`Carp`模块一起输出更为有用的诊断信息。对于我们现在的状况，可以用`longmess()`函数，因为它并非默认导入当前名称空间，所以需要显式声明：

```
use Carp qw(longmess);
```

调用`longmess()`可以返回`cluck()`生成的警告信息，较之传统的`warn()`函数，它还会回溯提供所有函数调用的堆栈详情，应该可以帮助你确定和诊断故障点。

发生故障的时间？

什么时候程序出错了。比方说，出错前最后读入的一行数据是什么？

为何给我邮件？

如果方便，尽可能预先回答读者心中的疑问：“为什么发邮件给我？”而回答可以只是简单一句“账户数据未能完整收集”，“DNS服务当前不可用”，或者“机房着火了”。收件者就会了解这封邮件的来由，偶尔也能促使他更积极地解决问题。

到底发生了什么？

最后，不要忘了在第一时间讲清楚发生的故障或错误。

这儿有一个简单的Perl程序，基本遵循了以上几个要点：

```
use Text::Wrap;
use Carp qw(longmess);
```

```

sub problemreport {

    # $shortcontext 应当是对问题的简单阐述，保持在一行以内
    # $usercontext 应当是对问题的详尽阐述
    # $nextstep 应当是解决问题的最佳建议
    my ( $shortcontext, $usercontext, $nextstep ) = @_ ;
    my ( $filename, $line, $subroutine ) = ( caller(1) )[ 1, 2, 3 ];
    my $report = '';

    $report .= "Problem with $filename: $shortcontext\n";
    $report .= "*** Problem report for $filename ***\n\n";
    $report .= fill( ' ', ' ', "- Problem: $usercontext" ) . "\n\n";
    $report
        .= "- Location: line $line of file $filename in " . "$subroutine\n\n";
    $report .= longmess('Stack trace ') . "\n";
    $report .= '- Occurred: ' . scalar localtime(time) . "\n\n";
    $report .= "- Next step: $nextstep\n";

    return $report;
}

sub fireperson {
    my $report = problemreport( 'the computer is on fire', <<EOR, <<EON);
    While running the accounting report, smoke started pouring out of the
    back of the machine. This occurred right after we processed the
    pension plan.
    EOR
    Please put fire out before continuing.
    EON

    print $report;
}

fireperson();

```

`problemreport()`输出完整的报告内容，第一行可以用作邮件主题，和前面的范例一样，我们可以用`Email::Send`发送该报告。这里的`fireperson()`函数用于测试它是否如期运行。

注意：最后一个小提示：如果打算写程序自动响应收到的邮件（如自动回复脚本），绝对有必要阅读RFC 3834, Recommendations for Automatic Responses to Electronic Mail，注意响应策略。

好了，关于发送邮件就是这么多内容，接下来我们看看怎样收取邮件。

收取邮件

用编程方式发送邮件的好处十分显见，但为何收取邮件也要傻傻地去编程呢？原因之一是可以借此来测试邮件服务器的工作状态。除了像往常那般测试邮件发送功能，我们还可以通过邮件协议测试其他各项功能。与其假扮邮件客户端软件只测试邮件收取功能，不如试着发送几份邮件然后再收取以测试完整的投递过程。当然，这么做可能不够完备，虽说每次发送与收取的邮件所走路线可能有所变化，但总比简单地测试邮件服务器是否打开socket端口监听连接要强。

就算没有自己的邮件服务器，仍然有好些理由该知道如何编程实现收取邮件。比方说，对ISP提供的垃圾邮件过滤功能不满意，可以编程获取所有邮件信息，用自己的过滤器依次检查并对垃圾邮件做上标记，以便后续阅读时跳过。当然，想要更完备的检查和析，也可以完整下载邮件后在本地处理。

使用POP3收取邮件

POP3（由RFC 1939描述）相对而言还算简单，我们先来看怎样用POP3协议收取邮件。以下是常见的POP3客户端与POP3服务器之间的交互步骤：

1. 连接并验证用户身份。
2. 查看是否有新邮件（此处还有更多细节）。
3. 请求返回最早的未读邮件内容并保存到本地计算机。
4. 请求服务器删除该邮件（服务器不会马上删除，只是先做上可以删除的标记）。
5. 重复第3步和第4步直到服务器上没有新邮件。
6. 发送结束信号并关闭连接。此时服务器才真正删除第4步被标注可以删除的邮件。

该协议非常简单，基本上也就上面这六个步骤了。^[注5]接下来，我们对这些操作的细节作一些解释。

首先有一个问题，如何判断当前服务器上有新邮件呢？假如第4步中每下载一封邮件就立即删除，那么答案显然非常简单：任何时候连接服务器时，上面的所有邮件必然都是新邮件。但事实上客户端并不需要每次下载后立即删除服务器上的备份，有时候留着备份还更有用。想想看，用户可能需要在不同地方（比如公司和家里的电脑）上阅读同一封

注5：这里我们省略了TOP命令这一步，因为在RFC中它是可选的。该命令请求服务器返回邮件头外加后续N行邮件正文。

邮件，这样的话，两处的邮件客户端一个只是下载阅读，另一个下载阅读后请求服务器删除。

没有要求服务器删除的客户端应该按某种办法记住已经阅读过的邮件，这一般利用RFC规定的可选命令UIDL来处理。UIDL要求服务器返回“唯一标识符清单”（unique-id listing），所以缩写为UIDL，服务器据此给每封邮件赋予一个唯一标识符，客户端则可以缓存该标识符，以便下次连接时仅下载标识符没有缓存的邮件。

有许多POP3模块可以使用，而且用起来还都非常简单。我用得最多的是Mail::POP3Client，因为它支持使用SSL连接（默认POP3使用明文发送密码验证用户身份）并且它所提供的方法名称基本和RFC 1939中的操作名称一致。说“基本”难免有吹毛求疵之虞，不过有时候它提供的方法名称虽然不同但更棒，比如Retrieve()（实际上这是HeadAndBody()的别名）在RFC 1939里面对应的方法名是RETR。我倒是期望所有模块的方法名称都遵照RFC来，哪怕只是做些别名也好。

下面的代码展示了如何建立安全连接、显示邮箱中的邮件数量、并打印第一封邮件内容：

```
use Mail::POP3Client;

my $pop3 = new Mail::POP3Client(
    USER      => 'user',
    PASSWORD   => 'secretsquirrel',
    HOST       => 'pop3.example.edu',
    USESSL     => 'true',
);

die 'Connection failed: ' . $pop3->Message() . "\n"
    if $pop3->Count() == ?1;

print 'Number of messages in this mailbox: ' . $pop3->Count() . "\n\n";
print "The first message looks like this: \n" . $pop3->Retrieve(1) . "\n";

$pop3->Close();
```

基本上没有多余代码，仅是利用提供的方法执行一下而已。如果要扩展功能，删除邮件可以调用Delete(message #)方法在服务器上标注为准备删除；记住邮件可以用Uidl()函数获取所有或特定邮件的唯一标识符。Head()和HeadAndBody()方法会根据调用上下文返回包含邮件内容的标量或数组。我们可以把邮件头和正文交给Mail::SpamAssassin来扫描判断是否为垃圾邮件，本章稍后会讨论该模块。

使用IMAP4rev1收取邮件

IMAP4rev1，现常称作IMAP4，是一种极为强劲而复杂的邮件操作协议，由RFC 3501描

述。它的基本理念和POP3截然不同。在POP3里，客户端负责周期性地发起请求并下载邮件，但IMAP4在整个阅读邮件的会话过程中都保持着持续连接。^[注6]使用POP3时，主要由客户端负责信息处理，像决断哪些邮件需要下载等。但IMAP4中客户端和服务端之间的信息交互要丰富得多，所以随之协议也必须更加智能化。到底有多智能？下面列出了IMAP4的一些特征：

- 能处理多级邮件文件夹及其内容。根据RFC 3051的说明：“IMAP4rev1支持创建、删除、重命名邮箱；检查新邮件、永久性删除邮件、设置和消除邮件标志位、可按RFC 2822和RFC 2045要求解析或搜索邮件、可根据邮件属性、正文文本等有选择地获取其中部分内容。”
- 能很好地理解邮件各部分结构。POP3只能返回整个邮件头或邮件头及后续若干行正文；而IMAP4则可以按要求仅返回“邮件正文的文本部分”，即跳过附件或HTML正文等数据量比较大的那些内容。因为在IMAP4里是按照标准MIME规范来解析邮件的。
- 允许客户端一次发送多条命令，并在随后以恰当方式返回响应内容。以往的标准交互都是客户端发送一条命令，然后等待服务器响应后再发送第二条命令，显然这么做效率低下。为了便于客户端和服务端都能分清整个会话过程的先后顺序，每个发送出去的IMAP4命令和响应内容都会打上前缀“标签”(tag)。
- 有一种“离线模式”(disconnected mode)，允许客户端连接到服务器后，一次缓存所有必要信息，完成之后断开。用户可以对缓存数据进行各项操作，就好像连接仍然保持时那样把命令发送至服务器。接下来再次连接时，客户端会把本地的所有历史操作到服务器上重演一遍，完成信息同步。这种离线模式特别适合乘坐飞机旅行的情况，在无网络可用时也能照常删除、过滤邮件，一旦回到网络环境，本地的所有变化会立即反映到IMAP服务器上。

功能越强，复杂度也就越高，开发成本也越大。手边没有RFC 3501参考的话，IMAP4编程会有一些难度。更何况IMAP服务器开发者根据自身理念不同，在实现功能时多少会有些侧重，所以开始编程时，最好循序渐进，从简单功能开始，逐步加入高级IMAP操作，慢慢完善程序。

对于接下来的代码范例，我打算用我自己喜欢的IMAP模块Mail::IMAPClient（最先由David J. Kernen编写，后由Mark Overmeer重写并维护）。从一台IMAP4服务器上迁移数据到另一台IMAP4服务器上的超级工具*imapsync*(<http://www.linux-france.org/prj/imapsync/dist/>)就是用该模块写成的。除了*imapsync*也选用它的原因之外，我还欣赏它直接发送IMAP命令的功能，这有时候真的很需要。其他可以考虑使用的模块有Rob

注6：警告：这种表述不够严谨，IMAP4有种被称为“离线模式”的说法，稍后我们会谈到。

Mueller写的Mail::IMAPTalk，他是Fastmail.fm网站的主要开发人员。虽说这个模块这几年来都没怎么更新，但他仍使我坚信现在的版本足够稳定，也一直经受着时间的考验。

我们的第一个代码范例，是要以安全方式连接到IMAP4服务器，访问用户邮箱，查找所有被SpamAssassin判为垃圾邮件（追加邮件头X-Spam-Flag: YES）并移至SPAM文件夹的邮件。下面先从连接代码开始：

```
use IO::Socket::SSL;
use Mail::IMAPClient;

my $s = IO::Socket::SSL->new(PeerAddr => 'imap.example.com',
                             PeerPort => '993',
                             Proto    => 'tcp');
die "$@" unless defined $s;

my $m = Mail::IMAPClient->new(User    => 'user', Socket=>$s,
                              Password=> 'topsecret');
```

Mail::IMAPClient和Mail::POP3Client一样都不支持SSL方式的加密安全连接，所以我们得自己动手构建一个加密的socket交给Mail::IMAPClient使用。如果不用Socket参数指定的话，所有通信内容，包括用户名和密码，都将以明文传输。

受限于旧版

如果你在用*imapsync*并且安装的Mail::IMAPClient还是老版本的话，当下可能会有些麻烦。在写本书时，*imapsync*尚未完全支持Mail::IMAPClient的3.x重写版本^[译注1]，所以上面的代码可能在你的机器上无法如期运行。如果要用自己的安全socket，需要修改上面的代码来增加额外的两项工作。

第一个问题，Mail::IMAPClient无法正确处理来自服务器的问候（greeting）信息。你得在创建socket后立即手动除掉，像这样：

```
my $greeting = <$s>;
my ( $id, $answer ) = split /\s+/, $greeting;
die "connect problem: $greeting" if $answer ne 'OK';
```

第二，Mail::IMAPClient无从获知连接是否已经建立，也就无法自动初始化登录操作，所以得在调用new()之后立即运行以下两行代码：

```
$m->State( Mail::IMAPClient::Connected() );
$m->login() or die 'login(): ' . $m->LastError();
```

译注1：目前最新的*imapsync* v1.311可以在Mail::IMAPClient v2.2.9或者至少在v3.0.019版本上运行。如果不是，*imapsync*会报错。

这两个问题在3.x版的Mail::IMAPClient中均已修复，希望将来它能和*imapsync*合作愉快。

STOP THE PRESSES:在本书付印之际，*imapsync*邮件列表传来消息，一系列宣称能解决各项兼容问题的补丁已经并入最新的Mail::IMAPClient模块。看来希望就在眼前，或许在你阅读此书时，这个问题已经不成为问题了。这个故事的寓意是：有时为了使用某个工具，不得已只能使用相应的旧版模块，眼馋新版功能却又不能升级。^[注7]

建立连接后的第一件事就是选择要操作的文件夹。这里我们希望打开用户的收件箱INBOX:

```
$m->select('INBOX');
```

现在搜索收件箱INBOX，查找邮件头X-Spam-Flag设置为YES的那些垃圾邮件:

```
my @spammsg = $m->search(qw(HEADER X-Spam-Flag YES));  
die $_ if @$_;
```

返回结果@spammsg是所有要移除的垃圾邮件，所以我们依次移动邮件到SPAM文件夹，然后关闭当前打开的文件夹，最后退出服务器:

```
foreach my $msg (@spammsg){  
    $m->move('SPAM', $msg) or die 'move failed: '.$m->LastError;  
}  
$m->close(); # 清理当前选中的文件夹并关闭  
$m->logout;
```

这里有个细节不得不提。之前我们谈到POP3的时候，说删除邮件之前先“标记为删除”的做法，在IMAP4里也是类似地分两步走：先将邮件标记为\Deleted，然后在关闭文件夹时删除。当我们要求移动邮件时，实际上是先复制邮件到新的文件夹，然后标记原来的邮件可删除。在此之前，我们可以直接用expunge()删除源文件夹中的废弃邮件，不过RFC 3501说在执行文件夹关闭操作(CLOSE)时会自动清理。

让我们另外再看一个例子，下节有关邮件处理方面的讨论也将由此展开。本节曾谈到IMAP4可以识别并处理邮件的MIME部分，下面的代码就展示了这方面的操作。为节省纸张少砍两棵树，我已经略去了模块加载、对象创建、安全连接、信箱选择等代码，它和上面例子差不多:

注7: Matt S. Trout 写的 local::lib 模块 (现由 Christopher Nehren 维护) 可让不同版本模块分立共存，并选择性使用特定版本。

```

my @digests = $m->search(qw(SUBJECT digest));

foreach my $msg (@digests) {

    my $struct = $m->get_bodystructure($msg);
    next unless defined $struct;

    # 邮箱中的每封邮件都会被赋予一个序列数和一个唯一标识。
    # 但默认 Mail::IMAPClient 仅使用邮件唯一标识符（即 UID）来区别不同邮件。
    print "Message with UID $msg (Content-type: ", $struct->bodytype, '/',
          $struct->bodysubtype,
          ") has this structure:\n\t",
          join("\n\t", $struct->parts) , "\n\n";
}

$m->logout;

```

以上代码查找当前选定文件中所有主题（Subject）行中包含“digest”字样的邮件，然后依次检查每封邮件的结构并打印MIME部分。下面是从我的收件箱*INBOX*中找到的两封邮件：

```

Message with UID 2457 (Content-type: TEXT/PLAIN) has this structure:
    HEAD
    1

Message with UID 29691 (Content-type: MULTIPART/MIXED) has this structure:
    1
    2
    3
    3.1
    3.1.HEAD
    3.1.1
    3.1.2
    3.2
    3.2.HEAD
    3.2.1
    3.2.2
    3.3
    3.3.HEAD
    3.3.1
    3.3.2
    4

```

可以看到每个MIME部分都有一个名称，我们可以调用**bodypart_string()**函数获取其内容，指定该邮件的UID及MIME部分的数字即可。比如这样：

```
print $m->bodypart_string(29691, '4');
```

打印UID为29691的邮件脚注部分的内容：

```

Perl-Win32-Database mailing list
Perl-Win32-Database@listserv.ActiveState.com

```

注意：Mail::IMAPClient是用Parse::RecDescent模块提取分离各个MIME部分的。多数时候，它的解析器运作良好，但偶尔碰上格式奇怪的邮件，它就会出乱子。所以，如果你现在要做的项目比较关注MIME处理的话，可以尝试专用于此的MIME解析模块，比如Email::MIME，甚至是之前提及的Mail::IMAPTalk模块。下节我们会看到一个使用Email::MIME的例子。

下一节，我们讨论邮件结构的展开和提取。

处理邮件

编程实现邮件收取只是第一步，本节我们来看看，对于收取下来的邮件都可以做哪些处理。

让我们从最基本的谈起，怎样剖析邮件结构以及拆解分析邮箱。还是老规矩，我们从Perl Email Project提供的模块开始。

注意：第一版中本节代码范例用的是Mail::Internet、Mail::Header和Mail::Folder模块。为保持本书前后的连贯性，我换用了Perl Email Project的模块，不过原来那些仍然是可以使用的（特别是Mark Overmeer正在不断定期更新前两个模块）。Mark同时也是Mail::Box的作者，该模块提供了极为丰富详尽的邮件及邮箱处理功能。如果觉得Perl Email Project的模块不尽其用，可以看看Mail::Box。

剖析单一邮件

Email::Simple模块提供了一种极为便捷的方法来提取RFC 2822兼容邮件的头信息。RFC 2822约定了邮件的格式，包括可接受邮件头的名字及其书写格式。

将整封邮件内容保存到标量变量，然后扔给Email::Simple就成了：

```
use Email::Simple;

my $message = <<'EOM';
From user@example.edu Mon Aug 6 05:43:22 2007
Received: from localhost (localhost [127.0.0.1])
    by zimbra.example.edu (Postfix) with ESMTp id 6A39577490A
    for <dnb@example.edu>; Mon, 6 Aug 2007 05:43:22 ?0400 (EDT)
Received: from zimbra.example.edu ([127.0.0.1])
    by localhost (zimbra.example.edu [127.0.0.1]) (amavisd-new, port 10024)
    with ESMTp id OIIgygSczEdt for <dnb@zimbra.example.edu>;
    Mon, 6 Aug 2007 05:43:22 ?0400 (EDT)
Received: from amber.example.edu (amber.example.edu [192.168.16.51])
```

```
by zimbra.example.edu (Postfix) with ESMTP id 2828A774909
for <dnb@zimbra.example.edu>; Mon, 6 Aug 2007 05:43:22 ?0400 (EDT)
Received: from chinese.example.edu ([192.168.16.212])
by amber.example.edu with esmtps (TLSv1:DHE-RSA-AES256-SHA:256)
(Exim 4.50)
id 1IHZA6-0002GV-7g
for dnb@example.edu; Mon, 06 Aug 2007 05:46:06 ?0400
Date: Mon, 6 Aug 2007 05:46:06 ?0400 (EDT)
From: My User <user@example.edu>
To: "David N. Blank-Edelman" <dnb@example.edu>
Subject: About mail server
Message-ID: <Pine.GS0.4.58.0708060544550.2793@chinese.example.edu>
```

Hi David,

Boy, that's a spiffy mail server you have there!

Best,

Your User
EOM

```
my $esimple = Email::Simple->new($message);
```

在所生成的`$esimple`对象上，你可以调用这两个方法：`header('field')`和`body()`。其中，`body()`方法会返回完整的邮件正文，这应该和你想的一样，但`header()`方法有点不同，它会根据给定的邮件头名称，在列表上下文中会返回所有该邮件头内容，在标量上下文中则返回第一个邮件头内容：

```
my @received      = $esimple->header('Received');
my $first_received = $esimple->header('Received');
```

`Email::Simple` 与其他邮件解析模块的不同处之一在于它仅返回对应邮件头的内容，而不是完整的包含邮件头名称的那一整行，比如：

```
print scalar $esimple->header('Date')
```

会打印：

```
Mon, 6 Aug 2007 05:46:06 ?0400 (EDT)
```

而不是：

```
Date: Mon, 6 Aug 2007 05:46:06 ?0400 (EDT)
```

不管出于怎样的需要，有时若要查看邮件中都有哪些邮件头，可以用`header_names()`返回。

解析邮件并不仅仅是提取邮件头或正文内容这么简单，还有许多额外事情要做。如

MIME编码的邮件，通常需要从邮件内容中提取附件部分并保存为另外一个不同的文件。下面是用Email::MIME实现的例子：

```
use Email::MIME;
use File::Slurp qw(slurp write_file);

my $message = slurp('mime.txt');

my $parsed = Email::MIME->new($message);

foreach my $part ($parsed->parts) {
    if ($part->content_type =~ /^application\/pdf;/i){
        write_file ($part->filename, $part->body);
    }
}
```

我们把邮件存为*mime.txt*，然后用*slurp()*一次读入所有内容，交给Email::MIME的*new()*方法创建对象；对象创建时，内部已经完成了对邮件的解析。接下来，依次循环每个MIME部分，根据其类型（content-type）判断是否为PDF格式文件。如果是，用该MIME部分中提供的文件名（如果没有提供，Email::MIME会自行生成一个文件名）保存其内容到一个文件。请注意，这里有两方面不够完备。其一，此代码仅查看第一层MIME部分中是否有附件。我们知道，MIME部分中还可以有子部分，如果有人转发某封邮件时将其作为附件，那么原始邮件中的附件就保存在第二层的MIME部分中，这里的代码无法提取。^[注8]其二，也是很严肃的问题，此代码信任邮件头中所给定的文件名，真实环境使用的程序必须严谨检查文件名所有字符是否有问题（请参阅后续补充内容的说明）。

解析邮件万不可掉以轻心

这里的提醒很重要，整个邮件解析处理过程都应注意这里提到的问题。要知道，解析邮件可是件很讲学问的事：一方面，邮件数据的复杂度可能出乎所料；另一方面，还有一伙坏人在存心混淆视听。所以我们的代码必须健壮完备，以应付各种非常规状况。

对于第一种情况，我这里有个例子：许多人都喜欢用简单的正则表达式验证电子邮件地址格式。不要和他们成为一路人。完整的RFC 2822规范约定的合法电子邮件地址格式是相当复杂的，我绝对可以保证你的代码总有一天要出问题，侥幸不

注8： 你可能想到用递归的方式逐层往下探查，在每个 *parts()* 里调用它的 *subparts()* 方法。不错，你可以用这个办法练练手。不过我想告诉你另外一条捷径，有个叫做 *Email::MIME::Attachment::Stripper* 的模块就是专门用来干这件事的。该模块也是由 Ricardo Signes 维护，它应该能帮到你。

得。相比之下，用现有的模块，比如Email::Valid（现由Ricardo Signes维护），或者Mail::Box中的Mail::Message::Field::*模块都能轻松解决问题。看看Regexp::Common模块（比如其中的Regexp::Common::net）是怎样解析地址的。类似这样的模块已经做了很多复杂数据处理工作，我们应该站在巨人肩膀上，直接使用这些完备的模块。此外，用Perl自己的-T开关（启用防污模式，即Taint模式），以防外部有所企图的数据破坏我们的程序的正常运行。

所谓坏人，多半是指那些发送垃圾邮件的人或组织。在反垃圾邮件领域拥有十多年经验的Bill Cole通过邮件告诉我说（当然，他允许我发表于此）：

垃圾邮件发送者无所不用其极，不管是故意捏造还是愚蠢失误，他们总是在制造各式诡异奇怪的邮件内容企图混过邮件过滤器。过滤垃圾邮件时绝不能相信对方宣称的一切，我们必须仔细武装邮件处理代码，即便前端已经有其他工具（比如MIMEDefang、sendmail或者其他工具）抵御了绝大部分，仍然有可能遇上漏网之鱼，尝试攻击我们的代码。

在结束本节之前，我还想举两个深入解析的例子。这回我们只是关注邮件正文。为保持叙述简洁，这里假设邮件正文为普通纯文本，和之前的例子一样未经任何编码。对于稍微复杂点的邮件，你完全可以用之前提到的模块转而得到最终的纯文本正文。

第一个例子会展示如何高效扫描正文来查找关键字。通常对于出现脏字眼的邮件，我们需要将其隔离起来。高效扫描邮件正文（特别是要扫描的词有一长串时）的关键是尽可能减少重复传递正文的次数，当然，最好一次都不要重复。是的，我们确实能做到：

```
my @dirty_words = qw ( sod ground soil earth filth mud shmutz );

foreach my $word (@dirty_words){
    return 'dirty' if ($body =~ /$word/is);
}
```

不过，这么做可就忙坏正则表达式引擎了。每当有脏字眼进来时都要重新解析/编译正则表达式，然后从头到尾扫描一遍邮件正文，如此往复费时费力。解决办法不止一种，最高效的是将脏字眼用管道符号（|）串起来写成一个正则表达式：

```
my $wordalt = join('|',@dirty_words);
my $regex = qr/$wordalt/is;

return 'dirty' if ($message =~ $regex);
```

这样就好些了，不过我们还可以走得更远。如果仔细查看这组脏字眼，你会发现它们的拼法有些共性，好些词都以同一个字母开头，所以我们可以据此优化正

则表达式来写成效率更高的`sod|il`)。Perl 5.10以上的正则表达式引擎会对这种写法进行内部性能优化。如果你用的是旧版且在乎性能，可以用Aaron Crane的`Text::Match::FastAlternatives`或David Landgren的`Regexp::Assemble`模块。对于这里的例子，前者构造的表达式比5.10以上的正则表达式优化引擎还要快些。`Regexp::Assemble`虽没那么快，但对于繁重的任务，它倒是有不少好功能可用。以下是`Text::Match::FastAlternatives`的简单例子：

```
use Text::Match::FastAlternatives;
use Email::Simple;
use File::Slurp qw(slurp);

my $message = slurp('message.txt');
my $esimple = Email::Simple->new($message);

my @dirty_words = qw ( sod ground soil earth filth mud shmutz );

# 清单越长，优化效果越明显
my $matcher = Text::Match::FastAlternatives->new( @dirty_words );

print 'dirty' if $matcher->match( $esimple->body() );
```

谈到`Text::Match::FastAlternatives`时有两个重要限制不得不说。其一，它仅能优化可打印的ASCII字符（所以如果是其他字符集就会出现問題）；其二，也是最要紧的限制，它只能按照区分大小写的方式执行匹配。如果我们想同时匹配小写的“filth”和首字母大写的“Filth”，我们不得不将上面代码的最后两行改写成：

```
my $matcher = Text::Match::FastAlternatives->new( map { lc } @dirty_words );

print 'dirty' if $matcher->match( lc $esimple->body() );
```

如果不在乎大小写，`Text::Match::FastAlternatives`确实可以为你的程序提高匹配速度。

以上只是查找关键字，如果要找更复杂的内容怎么办？有时从邮件提取并展开其中的URI会很有用。正如补充内容“解析邮件万不可掉以轻心”所谈及的，`Regexp::Common`模块就能很好地完成此项任务：

```
use File::Slurp qw(slurp);
use Email::Simple;
use Regexp::Common qw /URI/;

my $esimple = Email::Simple->new( scalar slurp $ARGV[0] );
my $body = $esimple->body;

while ( $body =~ /$RE{URI}{HTTP}{-keep}/g ) {
    print "$1\n";
}
```

该代码借用`Regexp::Common`里的正则表达式找出所有邮件消息中的URI。这里的`-keep`标志表示将匹配的URI内容保存到变量`$1`中。本章后续还会继续讨论与URI处理相关的更有趣的内容。

剖析整个邮箱

继续我们的话题，剖析整个邮箱，这部分说起来应该算比较简单直接的了。如果邮件存储在Unix上经典的`mbox`、`maildir`或者`mh`格式中，我们可以用来自Perl Email Project的`Email::Folder`模块（也是由Ricardo Signes维护）。很多常见的非Unix邮件代理，比如Eudora，也使用经典的Unix `mbox`格式存储本地邮件。所以该模块在各种平台上都能使用。

具体用法和之前的例子比较像，把表示邮箱的文件或目录扔给该模块就行了：

```
use Email::Folder;

my $folder = Email::Folder->new('FilenameOrDirectory');
```

构造方法`new()`接受文件名（适用于`mbox`格式的存储）或者目录名（适用于`maildir`和`mh`格式的存储）来存储邮件，返回`$folder`对象，用于表示包含一系列邮件的邮件文件夹。^[注9]由此取出的邮件用`Email::Simple`对象表示。我们可以一次取出所有邮件：

```
my @messages = $folder->messages;
```

或者一封一封取：

```
foreach my $message ($folder->next_message){
    ... # 某些针对邮件对象的操作
}
```

这里的`$message`实际上就是`Email::Simple`对象，所以可以直接用之前介绍过的方法。比如查看邮件主题`Subject`：

```
$subject = $message->header('Subject');
```

这些方法可以串起来写，所以下面的代码返回邮箱中下封邮件的`Subject`行：

```
$subject = $folder->next_message->header('Subject');
```

注9： `mbox`格式的邮箱将所有邮件保存在单一文件中；`maildir/mh` 则分别把每封邮件保存为独立文件，通过目录的方式组织。但无论哪种情况，`$folder`对象并不关心，具体邮件的存储方法都交由底层模块负责实现。

Email::Folder 本身只提供几种最基本的邮箱格式。^[注10]如果你需要更复杂的处理，可以用之前提到的Mail::Box模块。

再度抗争

至本书第二版更新素材时，有关垃圾邮件的章节的修订变得极具挑战性。到今天，垃圾邮件发送者和反垃圾邮件社区之间的斗争已经升级到了白热化的程度，原先第一版所讲的内容现在看来有些荒唐可笑，尽管在当时那些还算是好的建议。

以前，每位最终收到邮件的用户都可以挺身而出贡献一份力量，遇到垃圾邮件时向ISP投诉该邮件的发送来源地址。而如今，这种做法已经过时，成片的机器沦为傀儡，构成分布极广的僵尸网络（botnet），暗地发送着大量的垃圾邮件。数量上的悬殊差异，发送时机的快速变化，都令用户措手不及。要是还用以前的办法，简直就像在现代战争中举着长矛一样愚蠢。

说实话，在我最初开始重写本节时，我自己都不太确定哪些Perl工具会比较有用，直到最后摆上书架了也还是如此。我很难断言未来几年内两边的斗争会发展成什么样子。你看，自本书第一版以来就已经发生了那么多变化，天知道将来会怎样。但为了以飨读者，我求教了一些在反垃圾邮件领域工作多年且经验丰富的专家，他们给了我大量建议，其中大部分我都加工组织到“反垃圾邮件”一节中以及本章后续的章节里。希望这里介绍的最佳实践建议能帮到你。

反垃圾邮件

到目前为止，本章已经探讨了剖析信件及邮箱结构的一般工具，并看了一些应用程序，如从分离出来的邮件正文中扫描脏字眼等。而有关邮件处理技术的另一个应用领域（不幸的是恰好也是最大的领域）就是未经同意且大量发送的商业邮件，简单来说我们称之为“spam”。^[译注2]

SpamAssassin

正如补充内容所说，从任何角度看，防制垃圾邮件都已然成了一项棘手的工作，但为了保护自己的利益，我们必须竭尽全力与之斗争。这些年来，已经有许多人集聚了他们的智慧创造了各式软件，所以我们完全没必要从头开始。这儿有个Perl写的开源反垃圾

注10：有许多Email::Folder::模块支持其他各式邮箱，如在POP3、IMAP或Exchange服务器上指定的文件夹，解析过程就好和本地使用一样。虽然功能不多，都是些非常基本的，不过它们仍然很酷。

译注2：垃圾邮件称为spam，正常合法邮件称为ham。

邮件工具：Apache SpamAssassin(<http://spamassassin.apache.org>)，我们就从它入手。SpamAssassin的Perl API是由一系列Mail::SpamAssassin模块构成。该API在过去5年里都一直保持着稳定，所以未来几年也应该不会有什么大变化，尽可以放心使用。用此模块的另外一个原因是它提供了一些称手的处理邮件的函数，比如用于解码HTML或MIME格式的邮件、展开可读的URL地址、做一些黑名单查询等等不一而足。所以不管怎么说，该模块都值得我们学习一下。

和大多数看似神奇用来简易的Perl模块一样，Mail::SpamAssassin把最困难的事以最简洁的方式解决。想要知道SpamAssassin如何判断此邮件是不是垃圾邮件？不着急，这很容易。先看看基本用法：

```
use Mail::SpamAssassin;
use File::Slurp qw(slurp);

my $spama = Mail::SpamAssassin->new();
my $message = $spama->parse(scalar slurp 'message.txt');
my $status = $spama->check($message);

print (($status->is_spam()) ? 'spammy!' : "hammy!" . "\n");

$status->finish();
$message->finish();
```

上面这段代码在回答是否为垃圾邮件之前需要做三步处理：创建Mail::SpamAssassin对象，使用它来解析邮件内容至一个它可用的对象，最后调用check()方法进行检查。后面两步也可并作一步，直接调用check_message_text()即可，它一样可以解析并检查普通邮件的内容。但这么做有个缺点，我们无法得到表示邮件消息本身的对象，而之后我们有可能需要用到该对象，以便查询或处理原始邮件中的各个部分。来看看可以对该邮件对象做哪些操作吧。

首先是提取邮件RFC 2822相关的各个部分（比如说邮件头）的用法。例如，要获取所有Received头的列表，可以这么写：

```
use Mail::SpamAssassin;
use File::Slurp qw(slurp);

my $spama = Mail::SpamAssassin->new();
my $message = $spama->parse(scalar slurp 'message.txt');

my @received = $message->header('Received');

# 或者，仅仅获取最后[译注3]
```

译注3：每次邮件中转发时添加的Received头都放在邮件原文顶部，所以解析邮件获取的Received列表的最下方也就是最后的元素通常才是我们所关心的。原始邮件发送地址、连接SMTP时提供的HELO命令应答都在此Received头中，可用于之后垃圾邮件的判断。但Received头并非特例，所有header函数在标量上下文中返回的都是最后一个元素。


```
# 那个 Received 邮件头（注意，其他模块在标量上下文中给出的是列表的第一个
# 元素，而非最后一个。）
# my $received = $message->header('Received');

$message->finish();
```

也可以用Mail::SpamAssassin提取在邮件中找到的各个 MIME 部分。比如说，要打印所有 HTML 部分的内容，代码可以这么写：

```
use Mail::SpamAssassin;
use File::Slurp qw(slurp);

my $sa      = Mail::SpamAssassin->new();
my $message = $sa->parse( scalar slurp 'mime.txt' );
my @html_parts = $message->find_parts( qr(text/html), 1 );

foreach my $part (@html_parts) {
    print @{ $part->raw() };
}

$message->finish();
```

我们来看一下上面黑体展示的两行代码。其他几行之前已经介绍过，不再赘述。第一行调用了find_parts()方法，该方法会对邮件做一次完整的MIME结构解析，遍历各个层次的MIME部分，然后返回匹配正则表达式的MIME部分对象的引用（确切地说，是Mail::SpamAssassin::Message::Node对象）。该方法的第二个参数（1）表示仅返回第一层次的MIME部分。如果没有指定，find_parts()会返回所有层次上符合条件的MIME部分，所以可能会包含被转发邮件中的内容。

完整解析后要查看消息的所有部分（参阅补充内容“解析越少速度越快”），可以调用content_summary()方法。返回的内容类似这样：

```
DB<1> x $message->content_summary();
0 'multipart/mixed'
1 'multipart/alternative,text/plain,text/html'
2 'application/pdf'
```

解析越少速度越快

可能你会觉得有点奇怪，为什么Mail::SpamAssassin不在一开始解析时就做完整MIME扫描而非要等到某个方法运行时才去做？开始的那个parse()看起来很具有迷惑性，好像那时返回的对象就该是全部解析好的。当然不是！它仅是解析了邮件头而已。

所以，如果立即用content_summary()的话什么都看不到，必须先运行一次find_parts()方法来解析一遍才行。类似下面这样：

```
$message->find_parts(qr/./,1);
```

看起来有些另类的处理方式，实际上是为优化性能准备的。反垃圾邮件工具可不像普通的邮件解析模块，在实际运作的邮件服务器上，我们得让邮件快速通过Mail::SpamAssassin扫描，以保证大量邮件按时快速投递。所以为了提高性能，要尽可能减少对每封邮件非必要的开销。

一旦找到所需的MIME部分，可以和上面的范例一样，直接以原始格式打印，或者解码（比如用base64编码过的）、进一步规整化为文本（比如原来是HTML内容的）等等。

注意：现在是不是感觉有些似曾相识？之前在介绍Perl Email Project相关模块时我们也做过类似的事情，所以这不是你的幻觉，两组模块确实做了相同的事情。

那么，究竟该选用哪个模块呢？我个人倾向于按实际需求出发，不要混在一起用。如果只是一般的解析邮件内容，用不到Mail::SpamAssassin中有关垃圾邮件的部分，那就还是用Email::模块。如果确实是进行反垃圾邮件，或者对手上的邮件心存疑虑的话，就用Mail::SpamAssassin，哪怕只是想用它所提供的简便易用的函数（接下来就会看到）也行。

刚才我们看到，对通过parse()返回的Mail::SpamAssassin::Message对象可以执行各种方法来返回各类邮件信息，这就为各种类型的信息检测提供了方便。不再依赖于简单的check()方法，我们还可以结合各方面的蛛丝马迹综合判断一封邮件是否为垃圾邮件。

在邮件对象上运行check()或check_message_text()会返回Mail::SpamAssassin::PerMsgStatus对象^[译注4]。之前我们调用了\$status->is_spam()方法判断该对象是否为垃圾邮件，而实际上还有一些类似的检验方法。下面是部分我觉得比较常用的：

get_content_preview()

返回从邮件最开始几行摘录而成的短小文本，用作内容预览。

get_decoded_body_text_array()，**get_decoded_stripped_body_text_array()**

前者返回解码后的（比如base64编码过的）邮件正文，其他非文本MIME部分（如附件等）则被略去。后者（带...stripped...的版本）则会尝试将HTML格式的邮件正文转化为等效可读的纯文本内容后返回。

get_uri_list()，**get_uri_detail_list()**

前者提取所有在邮件中出现的URI并返回地址列表。后者（带...detail...的版本）则在解读每个URI后，以约定的数据结构详尽表述，合为列表返回。

译注4：后续代码中用\$status表示。

get()

和之前所用的header()方法非常相似，但用法不同，也更灵活。

让我们来仔细看看与URI相关的方法。

我们从邮件中展开所有URI是有原因的，因为URI具有指向性，所以可作为SpamAssassin判别垃圾邮件的依据。当垃圾邮件推销产品或服务时，总希望人们能到他们的网站上去购买，所以一般都会包含若干URL地址。通常这些地址不会出现在正常邮件中，所以借此可以判断其他包含该地址的邮件为垃圾邮件。甚至有时候不需要完整的URL，只需它的域名就足够了。SURBL黑名单(<http://www.surbl.org>)正是基于这种想法而来的（从已知垃圾邮件中提取发送者网站域名），效率非常高。“剖析单一邮件”一节获取URI清单的目的就在于此。

get_uri_detail_list()方法返回的数据结构如下所示（摘自模块文档）：

```
raw_uri => {
  types => { a => 1, img => 1, parsed => 1 },
  cleaned => [ canonified_uri ],
  anchor_text => [ "click here", "no click here" ],
  domains => { domain1 => 1, domain2 => 1 },
}
```

返回的清单是一个哈希的哈希，其中每个键为找到的原始URI地址，对应值为表述其详细内容的哈希。所以要取得所有URI中出现的域名列表，虽然有些复杂但也不算很难：

```
use Mail::SpamAssassin;
use File::Slurp qw(slurp);
use List::MoreUtils qw(uniq);

my $sa      = Mail::SpamAssassin->new();
my $status  = $sa->check_message_text( scalar slurp 'spam.txt' );
my $uris    = $status->get_uri_detail_list();

my @domains;

foreach my $uri ( keys %{$uris} ) {
  next if $uri =~ /^mailto:/;
  push( @domains, keys %{ $uris->{$uri}->{domains} } );
}

print join( "\n", uniq @domains );
```

取得域名清单后便可以依次查询它们是否被列入黑名单，比如常用的surbl.org。每个归入黑名单的域名，都会用一条主机名为该域名的surbl.org次级域名表示，实际上就是添加了一条DNS“A”记录。所以要查询某个域名是否在黑名单中，只需将此域名接上multi.surbl.org（比如makemoneyfast.com.multi.surbl.org），然后查询DNS记录，看返回

的主机名结果。查询DNS记录可以用第5章中演示过的`Net::DNS`模块。如果能够解析该主机名，就可以判定该邮件可能是垃圾邮件，因为其中有个链接的域名在黑名单中。

结束本节之前，让我们再来简要看下`get()`方法。要运行`get()`，必须首先用`check()`检查邮件对象，或用`check_message_text()`检查邮件内容。这两个方法都会返回一个表示邮件检查结果状态的对象^[译注5]，之后才可以在该对象上运行`get()`方法。`get()`能提取指定名称的邮件头信息，这点同`Email::Simple`的`header()`方法一样，但因为它是基于`check()`解析过的对象，所有另外还有些特殊用法，以便于提取更为具体的内容。比如，`header('From')`仅返回邮件的From头内容，但`get('From:addr')`却可以返回该邮件头中仅仅是电子邮件地址的那部分内容。同样，`get('From:name')`仅返回邮件地址中表示用户“名字”的那部分内容。如果原始邮件头内容如下：

```
David Blank-Edelman <dnb@example.edu>
```

那么用`:addr`得到的是“dnb@example.edu”，而用`:name`得到的是“David Blank-Edelman”。此外，`get()`方法还支持对某些虚拟邮件头的查询。虚拟邮件头实际是由其他头信息聚合而来的。比如要取得所有收件人地址（除Bcc以外），可以用`ToCc`作为虚拟邮件头名称。

在安装SpamAssassin时，我们可以配置“可信任的”和“不可信任的”主机名单。这样，本地网络服务器就可以和那些互联网上不明来历的危险服务器区分开来，据此判断是垃圾邮件的可能性。有若干`get()`支持的虚拟邮件头可返回与此相关的信息。不过我觉得最有意思的是X-Spam-Relays-Untrusted和X-Spam-Relays-Trusted这两个。下面是从真实垃圾邮件中摘取的Received头：

```
Received: from smtp.abac.com (smtp.abac.com [208.137.248.30])
    by amber.example.edu (8.8.6/8.8.6) with ESMTP id FAA29389
    for <user@ccs.example.edu>; Tue, 2 Dec 1997 05:51:56 ?0500 (EST)
Received: from smtp.abac.com (la-ppp-109.abac.com [209.60.248.109])
    by smtp.abac.com (8.8.7/8.8.7) with SMTP id CAA01384;
    Tue, 2 Dec 1997 02:53:33 ?0800 (PST)
Received: from mailhost.nowhere.com (alt1.nowhere.com (208.137.887.15))
    by nowhere.com (8.8.5/8.6.5) with SMTP id GAA00064 for <>;
    Tue, 02 Dec 1997 01:49:32 ?0600 (EST)
```

请求返回X-Spam-Relays-Untrusted头会得到（格式略作调整）：

```
[ ip=208.137.248.30 rdns=smtp.abac.com helo=smtp.
abac.com by=amber.example.edu ident= envfrom= intl=0 id=FAA29389 auth= msa=0 ]

[ ip=209.60.248.109 rdns=la-ppp-109.abac.com helo=smtp.abac.com
by=smtp.abac.com ident= envfrom= intl=0 id=CAA01384 auth= msa=0 ]
```

译注5：即`Mail::SpamAssassin::PerMsgStatus`对象。

这些邮件头经过SpamAssassin分析后，重组为容易辨识的格式，由此我们可以轻易看出其中可能存在的问题，比如中转服务器名字前后不一致、helo应答异常等等。上面例子的第二个Received头中，邮件发送者（在helo应答中）宣称自己来自smtp.abac.com，但实际上（很有可能）是从位于la-ppp-109.abac.com的拨号线路发出的^[译注6]。所以，像这种反常规行为，也可以作为判定垃圾邮件的依据。

反馈环路

有关垃圾邮件的讨论还有另外一面常常被人忽略。之前我们讨论的是收到邮件后如何评估是否为垃圾邮件，那么，反过来怎样才能防止别人把我们当作垃圾邮件发送者呢？

提供邮件营销服务的公司或个人与提供邮件系统托管服务的公司（特别是大型ISP），在某些理念上是一致的。他们都希望邮件用户能收到本该收到的邮件，同时对于恼人的垃圾邮件，不能让用户产生他们放任不管并染指其间的印象。发送营销邮件的人并不希望用户收到不感兴趣的广告，而邮件系统管理员也不希望用户因为看到了不想收到的邮件而愤怒。

正是出于这样的共同点，两个阵营的人开始相互协作。他们相互交换被判为垃圾邮件的相关信息。邮件系统运营商可以通过自己的软件判断垃圾邮件，也可以通过用户点击“此为垃圾邮件”按钮进行报告来判断。而发送营销邮件的人可以据此从自己的数据库中移除某些收件人地址等等。这个协作机制就称为反馈环路（feedback loops）。

某些大型邮件服务提供商允许邮件营销公司订阅反馈环路自动返回的报告邮件。一旦某封邮件被认为是垃圾邮件，邮件服务提供商就会发送一封邮件报告发送方。在写本书之际我所知道的最好的提供反馈环路服务的邮件服务提供商列在本章章末的“更多参考资料”一节的Spamhaus FAQ部分。

你可能会好奇Perl在其中扮演什么角色。经过长时间的摸索，人们最终制定了一种反馈报告的标准格式规范。该格式称为Abuse Reporting Format（ARF），定义具体细节规范的RFC见本章末的参考资料。它实际上是基于MIME的一种格式，这样可以方便自动化系统发送和接收。由于Perl本来就擅长于邮件解析，所以人们开始用Perl脚本自动解析邮件报告，然后再作适当后续处理。

你可能会说：“MIME? Great!”^[注11]我们之前刚刚学过解析MIME邮件，我知道该怎么做！”是的，这么说准错不了。基于标准的Perl MIME解析工具亲手写代码分析报告当然没什么问题，不过用现成的两类专门用于处理ARF报告的模块则会容易许多。接下来

译注6: rdns部分是从发送邮件来源IP地址反查DNS得到的域名，通常应该和helo应答部分一致。

注11: 或者，如果你和我一样对MIME爱恨有加，或许会选择另外一个词，以字母表中排在G前面的那个字母开头的词。

我会展示使用Email::ARF::Report解析ARF邮件的例子，该模块同样来自于Perl Email Project。如果需要发送ARF邮件，可以看看MIME::ARF模块，由Steve Atkins编写（下载地址位于<http://wordtothewise.com/resources/mimearf.html>）。

既然有了初步了解，那么接下来看看表8-1中典型的一封ARF邮件报告必须包含的三个MIME部分。表中所采用的范例取自ARF规范文档。

表8-1：ARF邮件结构

MIME部分	内容	范例
1	供人阅读的信息	This is an email abuse report for an email message received from IP 10.67.41.167 on Thu, 8 Mar 2005 14:00:00 EDT. For more information about this format please see http://www.mipassoc.org/arf/ .
2	供机器识别的元数据	Feedback-Type: abuse User-Agent: SomeGenerator/1.0 Version: 0.1
3	原始邮件的完整拷贝，包括邮件头	From: <somespammer@example.net> Received: from mailserver.example.net (mailserver.example.net [10.67.41.167]) by example.com with ESMTP id M63d4137594e46; Thu, 08 Mar 2005 14:00:00 ?0400 To: <Undisclosed Recipients> Subject: Earn money MIME-Version: 1.0 Content-type: text/plain Message-ID: 8787KJKJ3K4J3K4J3K4J3.mail@example.net Date: Thu, 02 Sep 2004 12:31:03 ?0500 Spam Spam Spam Spam Spam Spam Spam Spam Spam

有关范例邮件的上下文请参考ARF规范文档。用Email::ARF::Report模块分离各部分信息非常容易。下面是打印从ARF报告中提取的原始邮件部分信息的范例代码：

```
use Email::ARF::Report;  
use File::Slurp qw(slurp);  
  
my $message = slurp('arfsample1.txt');  
  
my $report = Email::ARF::Report->new($message);  
  
foreach my $header (qw(to date subject message-id)) {  
    print ucfirst $header . ': '  
      . $report->original_email->header($header) . "\n";  
}
```

如果觉得这段代码和之前看到的Email::Simple代码非常相近，这绝非巧合。Email::ARF::Report解析ARF邮件并提供返回各部分信息的方法。上面的范例中，我

们用`original_email()`方法访问原始邮件对象。而`original_email()`所返回的就是`Email::Simple`对象，所以延续之前学到的知识，我们可以继续调用`header()`方法打印头信息。只要我们能够提取报告中的信息，就能采取相应措施减少对用户造成的困扰（比如取消该用户的邮件订阅等等）。

有关垃圾邮件的话题就讨论至此，接下来换个轻松一点的话题，比如通过邮件和用户交互等等。

支持邮件的延展

就算在你的网站上没有“help desk”专栏，多少总能留个提供用户支持的邮件地址来解答各式问题。作为一种用户支持的沟通媒介，电子邮件有以下几点好处：

- 信息可以被存储，也能追踪进展，不像即时聊天。
- 可以错时交流。系统管理员可以用晚上的时间细致解答。
- 可以一对一、一对多地交流。如果14个人都有一样的问题，那么可以解决之后一起回复他们。
- 能方便转发给某个知晓问题答案的专家，由他负责答疑解惑。

这些都是在用户支持工作中不可或缺的需求，但使用电子邮件也有缺点：

- 如果邮件系统本身发生故障，或者用户发送邮件时碰到问题，必须切换到备用通信媒介。
- 用户可以在邮件中以任意格式输入任何内容，所以无法确保提供的信息足以判断并解决实际问题。甚至有时候根本不明白用户究竟想要表达什么。接下来我们试着看看如何解决这种难题。

一直以来我最喜欢举的例子就是下面这封完整的邮件，这里仅隐去了用户的真实姓名，免得他难为情：

```
Date: Sat, 28 Sep 1996 12:27:35 -0400 (EDT)
From: Special User <user@example.com>
To: systems@example.com
Subject: [Req. #9531] printer help
```

something is wrong and I have know idea what

如果没在邮件主题提及“printer”，我们根本无从入手，更不用说解决问题。诚然，这个例子有些极端。那么，看看更常见的吧：

```
From: Another user <user2@example.com>
Subject: [Req #14563] broken macine
```

To: systems@example.com
Date: Wed, 11 Mar 1998 10:59:42 -0500 (EST)

There is something wrong with the following machine:

krakatoa.example.com

用户肯定不是故意要发送这种缺乏上下文的邮件。我觉得产生这种现象的根本原因是用户和系统管理员对计算机系统的大脑认知模式有差异。

对多数用户来讲，可以看见的计算机系统仅限于所登录的终端机器，身边的打印机，还有可访问的数据存储资源（比如用户主目录）。而对系统管理员来说则完全不同，他知道有许多提供终端访问的服务器，并且还可能有若干附属的周边设备，而每台服务器都安装了不同的软件，运行状态也各有差异（如系统负载，软件配置等等）。

对用户来讲，问他“哪台机器有问题？”会很奇怪。他所说的计算机，不就是他正在用的那台么，如此明摆的事情为什么还要问？就像“帮我看看那台打印机”一样，系统管理员无从分辨所指的究竟是诸多打印机中的哪一台。

遇到具体问题的描述也是如此。每天收到类似“我的机器不工作了，你可以帮我看看吗？”的邮件，全世界的系统管理员都会咬牙切齿，耐着性子回信询问具体故障状况。因为一句简单的“不工作了”可以有无数种不同的表现症状，而每一种都各有其因。但对用户来讲，他所经历的一周内三次死机确实就是“不工作了”，他只是很自然地直接表述出来而已。

规定所发送邮件的内容或许是一种解决问题的办法，有些站点强制用户使用特定应用程序或网页表单提交故障描述。但很少有人愿意这么做，逐个字段填写，就只是为了提交一份报告或问一个简单的问题？这么麻烦，还不如不问呢。这和表单如何设计、是否漂亮毫无关系，如果没人愿意用它，它就没有实际存在的价值。很快，人们直接发来的邮件就会接踵而至，结果还是回到老路上。

好吧，有Perl来帮忙，情况或许不同。Perl可以重构收到的普通邮件，辅助我们改善技术支持的工作流程。首当其冲的是要确定问题的核心：“哪里有问题？哪台打印机？哪台机器？”等等。

我们先来看一个叫做*suss*的程序，它将演示如何用简洁的方式实现流程改进。该程序会根据邮件内容试着推测发生故障的机器名字。对于像“我的机器出问题了”这类邮件，不用回信询问用户，一般可以直接根据主机名来大致判断，继而排除疑难问题。

suss 使用的是一种非常简单的算法来猜测发生故障的主机名（基本上就是对邮件中的每个词进行哈希查找）。先查看邮件主题，然后搜索邮件正文，最后查看邮件头中最早的一个Received头。这里给出的代码是简化版本，仅是读取系统中的/etc/hosts文件以确定

可用主机名: [注12]

```
use Email::Simple;
use List::MoreUtils qw(uniq);
use File::Slurp qw(slurp);
my $localdomain = ".example.edu";

# 从主机文件中读取
open my $HOSTS, '<', '/etc/hosts' or die "Can't open hosts file\n";
my $machine;
my %machines;
while ( defined( $_ = <$HOSTS> ) ) {
    next if /^#/;          # 跳过注释行
    next if /^$/;          # 跳过空行
    next if /monitor/i;     # 一个令人误解的主机, 这里仅是例子

    $machine = lc( (split)[1] ); # 取第一列完整主机名并改为全小写
    $machine =~ s/\Q$localdomain\E//oi; # 去掉域名, 留下主机名
    $machines{$machine}++ unless $machines{$machine};
}
close $HOSTS;

# 解析邮件
my $message = new Email::Simple( scalar slurp( $ARGV[0] ) );

my @found;

# 检查邮件主题行
if ( @found = check_part( $message->header('Subject'), \%machines ) ) {
    print 'subject: ' . join( ' ', @found ) . "\n";
    exit;
}

# 检查邮件正文
if ( @found = check_part( $message->body, \%machines ) ) {
    print 'body: ' . join( ' ', @found ) . "\n";
    exit;
}

# 最后手段: 检查最早的一个 Received 行
my $received = ( reverse $message->header('Received') )[0];
$received =~ s/\Q$localdomain\E//g;
if ( @found = check_part( $received, \%machines ) ) {
    print 'received: ' . join( ' ', @found ) . "\n";
}

# 根据给定邮件的不同部分, 查找所有出现在主机查找表中的名字
sub check_part {
    my $part      = shift; # 不同部分的文本内容
    my $machines = shift; # 对主机查找表的引用

    $part =~ s/[\w\s]//g;
    $part =~ s/\n/ /g;
```

注12: 实际使用时, 可以考虑改用更好的主机查找方式, 比如缓存 DNS 区域传送的一份拷贝, 或者遍历一次 LDAP 树。

```

    return uniq grep { exists $machines->{$_} } split( ' ', lc $part );
}

```

还有一点需要说明：如果像“My monitor is broken.”这样的句子出现在邮件中，而*monitor*这个名字又碰巧用在某台服务器上，那么上面代码中一带而过的单词检查可能会产生问题。所以，如果出现类似状况，就需要作特别处理，要么和我们一样简单地以`next if /monitor/i;`跳过，要么改进程序的解析方案（scheme）。

暂且放下代码，我们来看看两封真实的用户邮件：

```

Received: from strontium.example.com (strontium.example.com [192.168.1.114])
    by mailhub.example.com (8.8.4/8.7.3) with ESMTP id RAA27043
    for <systems>; Thu, 29 Mar 2007 17:07:44 ?0500 (EST)
From: User Person <user@example.com>
Received: (user@localhost)
    by strontium.example.com (8.8.4/8.6.4) id RAA10500
    for systems; Thu, 29 Mar 2007 17:07:41 ?0500 (EST)
Message-Id: <199703272207.RAA10500@strontium.example.com>
Subject: [Req #11509] Monitor
To: systems@example.com
Date: Thu, 29 Mar 2007 17:07:40 ?0500 (EST)

```

```

Hi,
My monitor is flickering a little bit and it is tiresome
whe working with it to much.
Is it possible to fix it or changing the monitor?

```

Thanks.

User.

```

-----
Received: from example.com (user2@example.com [192.168.1.7])
    by mailhost.example.com (8.8.4/8.7.3) with SMTP id SAA00732
    for <systems@example.com>; Thu, 29 Mar 2007 18:34:54 ?0500 (EST)
Date: Thu, 29 Mar 2007 18:34:54 ?0500 (EST)
From: Another User <user2@example.com>
To: systems@example.com
Subject: [Req #11510] problems with two computers
Message-Id: <Pine.SUN.3.95.970327183117.23440A-100000@example.com>

```

```

In Jenolen (in room 292), there is a piece of a disk stuck in it. In intrepid,
there is a disk with no cover (or whatever you call that silver thing) stuck in
it. We tried to turn off intrepid, but it wouldn't work. We (the proctor on duty
and I) tried to get the disk piece out, but it didn't work. The proctor in charge
decided to put signs on them saying 'out of order'

```

AnotherUser

运行我们的程序，针对两封邮件会分别有如下输出：

```
received: strontium
```

及：

```
body: jenolen intrepid
```

可以看到，所有的主机名猜测都正确，而我们只是用了一小段简单的代码就实现了。更进一步，假设我们收到下面这封邮件，该用户全然不知我们总共有30台打印机：

```
Received: from [192.168.1.118] (buggypeak.example.com [192.168.1.118])
        by mailhost.example.com (8.8.6/8.8.6) with SMTP id JAA16638
        for <systems>; Tue, 7 Aug 2007 09:07:15 ?0400 (EDT)
Message-Id: <v02130502b1ecb78576a9@[192.168.1.118]>
Date: Tue, 7 Aug 2007 09:07:16 ?0400
To: systems@example.com
From: user@example.com (Nice User)
Subject: [Req #15746] printer

Could someone please persuade my printer to behave and print like a nice printer
should? Thanks much :)

-Nice User.
```

难不倒我们。有Perl帮忙，我们大致也可以推测出发生故障的打印机。一般用户都会选择离自己座位最近的那台打印机，所以只要知道用户发送邮件时所在的机器，再根据网络拓扑图，便能推算出发生故障的打印机。获取终端机器到打印机映射关系的办法有许多（从独立的文件查询，或者通过第5章中提及的主机数据库查询，甚至通过LDAP目录服务查询）。下面的代码使用了最简单的主机名到相关打印机映射关系数据库：

```
use Email::Simple;
use File::Slurp qw(slurp);
use DB_File;

my $localdomain = '.example.com';
my $printdb     = 'printdb';

# 解析邮件
my $message = new Email::Simple( scalar slurp $ARGV[0] );

# 检查邮件主题行
my $subject = $message->header('Subject');

if ( $subject =~ /print(er)?/i ) {

    # 找出发送邮件的主机
    my $received = ( reverse $message->header('Received') )[0];
    my ($host) = $received =~ /\((\S+)\Q$localdomain\E \[/;

    tie my %printdb, 'DB_File', $printdb
        or die "Can't tie $printdb database: $!\n";

    print "Problem on $host may be with printer " . $printdb{$host} . ".\n";

    untie %printdb;
}
}
```

如果邮件在主题行中提到“print”、“printer”或是“printing”，就从邮件最早的Received头中提取发送邮件主机的名字。因为我们要知道我们的邮件服务器所用的Received头的记录格式，所以据此构造一条正则表达式来提取其中的主机名字字符串（如果你的邮件投递服务器记录格式与此不同，则要修改这里的正则表达式）。然后，只需到Berkeley DB数据库中查找关联的打印机即可。最终输出：

```
Problem on buggypeak may be with the printer called prints-charming.
```

静下心来观察你所在的工作环境，应该会有更多更具体的改良技术支持邮件的办法。上面的例子非常简单，只是抛砖引玉，希望对你有所启发。

可以根据你的具体情况修改之前的`suss`程序以作为各种票单系统的前端处理环节。结合我们刚刚学过的获取并解析邮件的知识，你可以让用户统一把邮件发送到单个“catchall”地址，比如“helpdesk@example.com”，然后再由程序自动分析。如果对提问主题的确定的话（像Mail::SpamAssassin一样通过规则分值累计，最终用于确定有十足把握），便可自动转寄邮件到负责处理该问题的人。

Perl 提供了各种分析邮件的办法，大大小小的应用场景都有其用武之地。作为练习，请思考一下你身边可能有用的那些读取邮件（比如说由其他程序自动发送的邮件）的辅助程序应该怎么实现。

本章所用模块

模块名	CPAN ID	版本
Mac::Carbon	CNANDOR	0.77
Win32::OLE（随 ActiveState Perl 发布）	JDB	0.1709
Mail::Outlook（位于 MailTools 模块内）	BARBIE	0.13
Email::Send	RJBS	2.192
Email::Simple	RJBS	2.003
Email::Simple::Creator	RJBS	1.424
Email::MIME	RJBS	1.861
Email::MIME::Creator	RJBS	1.454
File::Slurp	DROLSKY	9999.13
Email::MIME::CreateHTML	BBC	1.026
Text::Wrap（位于 Text-Tabs+Wrap 模块内，随 Perl 发布）	MUIR	2006.1117
File::Spec（位于 PathTools 模块内，随 Perl 发布）	KWILLIAMS	3.2701
IO::Socket（位于 IO 模块内，随 Perl 发布）	GBARR	1.30
Carp（随 Perl 发布）		1.08
Mail::POP3Client	SDOWD	2.18

模块名	CPAN ID	版本
Mail::IMAPClient	MARKOV	3.08
Mail::IMAPTalk	ROBM	1.03
IO::Socket::SSL	SULLR	1.13
Mail::Box	MARKOV	2.082
Text::Match::FastAlternatives	ARC	1.00
Regexp::Common	ABIGAIL	2.122
Email::Folder	RJBS	0.854
Mail::SpamAssassin	JMASON	3.24
List::MoreUtils	VPARSEVAL	0.22
Email::ARF::Report	RJBS	3.01
DB_File (随Perl发布)	PMQS	1.817

更多参考资料

本章中的POP3和IMAPv4小节修订自我原本为USENIX协会的《;login》杂志2008年2月号撰写的专栏。

《Network Programming with Perl》，由Lincoln Stein所著（Addison-Wesley出版），它是关于用Perl进行网络服务器编程的最好的书籍之一。

《Perl Cookbook》，由Tom Christiansen和Nathan Torkington所著（O'Reilly出版），也涉及网络服务器编程。

<http://www.cauce.org>是反未经许可商业邮件联盟(Coalition Against Unsolicited Commercial Email)的网站。有很多致力于反垃圾邮件的网站，这个网站是个很好的起点，它拥有很多指向相关网站的链接，包括对邮件头进行深入分析的内容。

<http://emailproject.perl.org>是Perl电子邮件项目(Perl Email Project)的主页。

<http://www.spamhaus.org/faq/>有很多非常好的关于反垃圾邮件的常见问题(FAQ)列表，其中就包括了互联网服务提供商处理反馈环路垃圾邮件的主题，以及如何处理其客户所面临（或造成）的垃圾邮件问题。

<http://wordtothewise.com/resources/arf.html>和<http://mipassoc.org/arf/index.html>是两个很好的关于ARF标准的信息资源。关于这个标准自身最新的（本书写作之时）草案可以在<http://www.ietf.org/internet-drafts/draft-shafranovich-feedback-report-07.txt>找到。（最新的草案内容请参考<http://www.ietf.org/internet-drafts/>。）

如果你想从服务器角度尝试大容量邮件的处理（尤其是为了反垃圾邮件），那么有两个

非常有趣的软件可能是你想了解的：*qpsmtpd* (<http://smtpd.develooper.com>) 和 *Traffic Control* (<http://www.matlchannels.com>)。前者是开源的软件包，后者是在某些条件下可以免费使用的商业软件包。他们都是用Perl写的SMTP处理器/守护进程，都是被安置在标准MTA之前用来只对合法邮件进行代理的。对本章来说，这两个软件之所以有趣是因它们的插件功能。用户可以通过Perl编写插件以改变或决定通过这些软件包处理的邮件会被如何处理。通常这些插件会做一些类似垃圾邮件/正常邮件测定的工作，但说真的，创意是可以无穷的。

你也许还想看看下面提到的RFC文档：

- *RFC 1939: Post Office Protocol* (第3版)，由J. Myers和M. Rose 所著（1996年）
- *RFC 2045: Multipurpose Internet Mail Extensions (MIME) Part One: Format of Internet Message Bodies*，由N. Freed和N. Borenstein所著（1996年）
- *RFC 2046: Multipurpose Internet Mail Extensions (MIME) Part Two: Media Types*，由N. Freed和N. Borenstein所著（1996年）
- *RFC 2047: MIME (Multipurpose Internet Mail Extensions) Part Three: Message Header Extensions for Non-ASCII Text*，由K. Moore所著（1996年）
- *RFC 2077: The Model Primary Content Type for Multipurpose Internet Mail Extensions*，由S. Nelson和C. Parks所著（1997年）
- *RFC 2821: Simple Mail Transfer Protocol*，由J. Klensin所著（2001年）
- *RFC 2822: Internet Message Format*，由P. Resnick所著（2001）
- *RFC 3501: INTERNET MESSAGE ACCESS PROTOCOL* (第4版第1次修订)，由M. Crispin所著（2003年）
- *RFC 3834: Recommendations for Automatic Responses to Electronic Mail*，由K. Harrenstien和K. Moore所著（2004年）
- *RFC 4288: Media Type Specifications and Registration Procedures*，由N. Freed和J. Klensin所著（2005）
- *RFC 4289: Multipurpose Internet Mail Extensions (MIME) Part Four: Registration Procedures*，由N. Freed和J. Klensin所著（2005）

目录服务

信息系统的规模越大，越难以找到信息，有时甚至让人感觉完全没法用。随着网络越来越复杂，目录服务会渐渐成为必需。网络用户需要使用目录服务来找到其他用户，以便给对方发邮件（或共享别的信息）。目录服务也用来在网络中广播某些可用资源，比如打印机或网络上的共享磁盘。公共密钥验证系统也可以使用目录服务来发布信息。在这一章我们会分享如何使用Perl来与某些常见的目录服务交互，包括Finger、WHOIS、LDAP和Active Directory（通过Active Directory Service Interface）。

什么是目录？

在第7章，我曾经提到过几乎所有的系统管理都是在与某种数据库打交道，而目录就相当于某种类型的数据库。这里我们通过对比“数据库”和“目录”来认识目录的主要特性：

网络化

目录几乎总是在网络中应用的。数据库可能会与其客户端程序在同一台机器上（如 `/etc/passwd` 文件），而目录服务离开网络几乎没有任何意义。

简单通信/数据操纵

数据库几乎总是使用比较复杂的查询语言来进行数据查询和操纵。我们在第7章SQL（以及附录D）介绍了其中最普及的SQL。相比之下，同目录服务打交道的工作就容易多了。目录客户端一般只执行初步操作并且不使用功能完备的语言作为它与服务器通信的一部分。所以，进行任何查询一般都较简单。

层次性

如今的目录服务大多建议采用树形的信息结构，而数据库大多不是这样的。

如今的目录服务是为了特殊类型的数据流量而优化的。在正常情况下，目录查询（读）的频率远远高于修改（写）的频率。

如果你遇到的信息系统看上去像数据库（可能真的是有数据库后端）但是有以上的特性，那么你可能已经在和目录服务打交道了。在接下来我们将要介绍的四中目录服务中，这些特性应该是比较容易识别的。

Finger：一个简单目录服务

Finger和WHOIS是简单目录服务的典型例子。Finger主要是为了提供某台机器的用户信息而存在的（当然我们会展现一些使用它来实现的创意服务）。不过后来更高版本的Finger（比如GNU Finger及其衍生版本）提供了更高级的功能，允许通过一台服务器查询网络中所有主机的信息。

又拿石头砸自己脚？

我觉得如今你可能很难找到提供Finger的站点了，互联网的存在以及安全性的顾虑导致它几乎绝迹。那为什么本书还要介绍它呢？

这里引入Finger的简介有一个原因：它非常好学。这个协议非常简单，它成为了那些基于文本的网络服务的最佳选择，而且这些网络服务也不必担心缺少模块支持。如果你碰巧遇到通过这个服务发布的信息，那么应该会庆幸曾经学过这个老掉牙的协议。

Finger是最早广泛使用的目录服务。人们曾经广泛使用它的客户端finger命令来查询别的网站的用户的邮件地址（有时甚至用来查本网站的用户）。`finger harry@hogwarts.edu`会告诉你到底Harry的邮件地址是harry还是hpotter，或者是别的什么词（因为它会一并列出学校里面所有的Harry）。Finger的普及后来逐渐衰退，因为设置网站主页变得更加普及，而且如此轻易泄露用户信息也被认为不负责任。

从Perl里面获取Finger协议的信息是“条条大路通罗马”这一座右铭的又一见证。2000年我第一次在CPAN搜索这个模块的时候还没有什么发现，但如今你再搜索就会发现Dennis Taylor的Net::Finger模块，它发布于我第一次搜索的半年之后。稍后你就会看到它的使用方法，不过目前我们先假定它并不存在，而是从头实现它。这展示了在“特效”模块没有出现时，该如何使用通用模块实现某种协议。

Finger协议本身是一个非常简单的基于文本的TCP/IP，定义在RFC 1288中。客户端以

TCP访问服务器的79号端口，然后发送一个简单的回车换行结束的^[注1]字符串。这个字符串既可以包含要查询的客户端，也可以是空的（这意味着要求查询所有的用户）。服务器在发送响应数据之后关闭连接。你可以使用telnet连接到远程主机的Finger端口，输入下面的命令来显示交互过程^[注2]：

```
$ telnet quake.geo.berkeley.edu 79
Trying 136.177.20.1...
Connected to gldfs.cr.usgs.gov.
Escape character is '^]'.
/W quake<CR><LF>

      RAPID EARTHQUAKE LOCATION SERVICE
      U.S. Geological Survey, Menlo Park, California.
      U.C. Berkeley Seismological Laboratory, Berkeley, California.
      (members of the Council of the National Seismic System)

...
DATE-(UTC)-TIME  LAT   LON   DEP  MAG  Q  COMMENTS
yy/mm/dd hh:mm:ss  deg.  deg.  km
-----
09/01/12 16:29:37  36.03N 120.59W  4.5 2.3Md A*  20 km NW of Parkfield, CA
09/01/13 08:17:38  38.81N 122.82W  2.4 2.1Md A*   2 km NNW of The Geysers, CA
09/01/13 11:51:09  40.66N 124.04W 23.6 2.5Md B*  12 km NE of Fortuna, CA
09/01/13 18:27:01  36.80N 121.51W  5.5 2.4Md A*   5 km SSE of San Juan Bautista, CA
09/01/14 00:29:11  39.37N 123.27W  3.1 2.2Md B*   8 km ESE of Willits, CA
09/01/14 01:48:23  38.24N 118.69W 12.0 2.3Md C*  17 km WSW of Qualeys Camp, NV
09/01/14 02:06:57  38.24N 118.69W  6.0 2.2Md C*  17 km WSW of Qualeys Camp, NV
09/01/14 03:44:02  38.82N 122.83W  2.1 2.1Md A*   3 km NW of The Geysers, CA
09/01/14 05:08:21  36.74N 121.34W  9.1 3.4Ml A*   6 km SSW of Tres Pinos, CA
09/01/14 07:46:02  39.04N 123.34W  0.1 2.2Md C*  17 km SW of Ukiah, CA
09/01/14 10:24:53  40.42N 125.07W  1.2 3.1Ml C*  67 km W of Petrolia, CA
09/01/14 17:32:54  38.84N 122.83W  1.9 2.2Md A*   5 km NNW of The Geysers, CA
09/01/14 17:57:34  36.56N 121.16W  6.7 2.4Md A*   4 km NNW of Pinnacles, CA

...
$
```

在这里我们直接登录到了quake.geo.berkeley.edu机器的Finger端口。我们输入“quake”作为用户名（使用/W来获取详细信息），然后服务器返回了关于这个用户的信息。

我选择这个特殊的主机和用户是为了展示（在互联网的早期）如何使用Finger服务器来发布一些有趣的信息。你可以用Finger服务器来查询自动贩卖机、热水器或其他某种传感装置。刚才的例子展示了如何用Finger查询地震仪记录的信息。

不幸的是，通过Finger发布这些有趣的信息已经越来越罕见了。到目前为止，Bennet Yee收集的“Internet Accessible Coke Machines”和“Internet Accessible Machines”页面

注1： 回车+换行，即ASCII 13 + ASCII 10。

注2： 这些Finger服务器曾经工作过。这只是我测试过的16个服务器中的一个。如果你没有从服务器获得上面的信息，请相信我它确实曾经返回过这些信息。

(<http://www.bennettye.org/ucsd-pages/fun.html>)中列出的Finger主机已经完全停止工作了。就这种类型的任务而言, HTTP已经完全代替了Finger服务。互联网中肯定还有Finger服务器存在, 不过应该主要是供内部使用了。

尽管如今Finger服务器已经不那么常见了, 但它的简单仍能使初学者受益。让我们用Perl来实现刚才用telnet命令实现的交互过程。在Perl里面我们应该仍然需要通过网络socket完成通信, 但不必使用底层的socket命令, 因为我们有Jay Rogers写的Net::Telnet模块^[注3]。

Net::Telnet将为我们处理所有连接设置工作并且提供用于经此连接发送和接收数据的简单接口。虽然我们这个例子不会使用它的全部功能, 但Net::Telnet仍提供了一些便利的模式扫描机制, 以允许程序监听来自其他服务的特定响应。

下面的代码是Net::Telnet版本的Finger客户端。这段代码能接受user@finger_server这种格式的参数。如果省略了用户名, 那么就返回服务器上所有的活跃用户。如果主机名被省略, 那么就查询本地主机:

```
use Net::Telnet;

my($username,$host) = split(/\@/, $ARGV[0]);
$host = $host ? $host : 'localhost';

# 创建新连接
my $cn = new Net::Telnet(Host => $host,
                        Port => 'finger');

# 在该连接中发送用户名
# /W 是 RFC 1288 定义的命令, 表示回显详细信息
unless ($cn->print("/W $username")){
    $cn->close;
    die 'Unable to send finger string: '.$cn->errmg."\n";
}

# 抓取所有收到的数据, 直到连接结束
my ($ret,$data);
while (defined ($ret = $cn->get)) {
    $data .= $ret;
}

# 关闭连接
$cn->close;

# 显示收到的数据
print $data;
```

注3: 如果Net::Telnet没有让你满意, 还可以使用Austin Schutz的(目前维护者是Roland Giersig) Expect.pm模块来驱动telnet或者其他网络客户端。

你可能注意到了传给`print()`的字符串中的`/W`。RFC 1288声明`/W`开关能放在用户名前面，让服务器输出更加详细的用户信息。

如果你需要连接其他类似Finger的基于TCP的文本协议，那么可以采用类似的代码。比如下面的代码可以连接到daytime服务器（显示主机的当前时间）：

```
use Net::Telnet;

my $host = $ARGV[0] ? $ARGV[0] : 'localhost';

my $cn = new Net::Telnet(Host => $host,
                        Port => 'daytime'); port 13

my ($ret,$data);
while (defined ($ret = $cn->get)) {
    $data .= $ret;
}
$cn->close;

print $data;
```

现在你应该体会到基于TCP的客户端是多么容易创建了。但如果已经有了专门处理某个协议的模块，那么事情还会更加简单。对于Finger来说，你可以使用Taylor的`Net::Finger`模块来把整个任务变成一个函数调用：

```
use Net::Finger;

# finger() 函数接受类似 user@host 这样的字符串作为参数，然后返回收到的数据
print finger($ARGV[0]);
```

为了展示所有可行方案，我还要补充一下，其实也可以通过调用外部可执行程序的方法来实现：

```
my($username,$host) = split('@',$ARGV[0]);
$host = $host ? $host : 'localhost';

# finger可执行程序的位置
my $fingerex = ($^O eq 'MSWin32') ?
    $ENV{'SYSTEMROOT'}.'\\System32\\finger' :
    '/usr/bin/finger'; # 也可能是 /usr/ucb/finger

print ` $fingerex ${username}@${host} `
```

现在你看到了如何使用三种不同的方法来执行Finger请求。最后一种方法应该是最不理想的，因为它需要调用另外一个程序。`Net::Finger`能简化Finger请求；而对大多数非标准协议来说，`Net::Telnet`或类似方法则更容易达到要求。

WHOIS目录服务

WHOIS是另一种常见的只读目录服务。WHOIS提供了一种类似电话目录的服务，其中列出了主机、网站和运行网站的人。有些大型机构（如IBM、UC Berkeley和MIT）有自己的WHOIS服务，但最重要的WHOIS服务器是由InterNIC和其他互联网注册机构运营的，比如RIPE（European IP address allocations）和APNIC（Asia/Pacific address allocations）。

如果你必须要和某个网站的系统管理员联系以汇报可疑网络活动，那么可以用WHOIS来获取联系信息^[注4]。在大多数操作系统上都有图形用户界面或者命令行工具来查询WHOIS信息。所有的注册机构也都有基于Web的WHOIS查询页面。在Unix平台上，查询WHOIS信息的命令行界面是这样的：

```
% whois -h whois.educause.net brandeis.edu
<instructional paragraph omitted>
Registrant:
  Brandeis University
  Library and Technology Services MS017
  415 South Street
  Waltham, MA 02453-2728
  UNITED STATES

Administrative Contact:
  Director for Networks & Systems
  Brandeis University
  Library and Technology Services
  MS017 PO Box 9110
  Waltham, MA 02454-9110
  UNITED STATES
  (781) 736-4569
  noc@brandeis.edu

Technical Contact:
  NetSys
  Brandeis University
  Library and Technology Services
  MS017 PO Box 9110
  Waltham, MA 02454-9110
  UNITED STATES
  (781) 736-4571
  hostmaster@brandeis.edu

Name Servers:
  LILITH.UNET.BRANDEIS.EDU      129.64.99.12
  FRASIER.UNET.BRANDEIS.EDU    129.64.99.11
  NS1.UMASS.EDU
```

注4： 如果你读了上面那句话之后觉得有点晦涩，那是因为这句简单的话背后有很多需要解释的地方。在一两页之后你应该会明白什么情况下需要向系统管理员汇报问题。

NS2.UMASS.EDU
NS3.UMASS.EDU

Domain record activated: 27-May-1987
Domain record last updated: 11-Jun-2008
Domain expires: 31-Jul-2009

如果你想了解某个IP地址范围的所有者，WHOIS也是很好的工具：

```
% whois -h whois.arin.net 129.64.2
OrgName:  Brandeis University
OrgID:    BRANDE
Address:  415 South Street
City:     Waltham
StateProv: MA
PostalCode: 02454
Country:  US

NetRange: 129.64.0.0 - 129.64.255.255
CIDR:     129.64.0.0/16
NetName:  BRANDEIS
NetHandle: NET-129-64-0-0-1
Parent:   NET-129-0-0-0-0
NetType:  Direct Assignment
NameServer: LILITH.UNET.BRANDEIS.EDU
NameServer: FRASIER.UNET.BRANDEIS.EDU
Comment:
RegDate:  1987-09-04
Updated:   2002-10-24

TechHandle: ZB114-ARIN
TechName:  Brandeis University Information Technology
TechPhone: +1-781-736-4800
TechEmail: hostmaster@brandeis.edu

# ARIN WHOIS database, last updated 2009-01-13 19:10
# Enter ? for additional hints on searching ARIN's WHOIS database.
```

上面展示了Unix和Mac OS X平台有以使用的命令行WHOIS客户端。Windows平台没有预装这样的客户端，不过这并不是问题。有很多自由软件或者共享软件可以使用，比如cygwin发行版就自带了一个，稍后要介绍的Net::Whois::Raw模块也提供了一个很好的客户端。

刚才说到有需要解释的地方，现在我们就兑现承诺，解释具体的情况。目前看来，WHOIS随着互联网的发展变得难以捉摸。坦率地说，之前能够查询WHOIS信息的几个Perl实现已经变得不太可靠了。

让我尝试尽可能简单地说明问题。曾经所有与互联网相关的WHOIS信息只有一个注册机构，这使得用Perl代码来查询并分析结果非常容易。后来因为政治或者技术方面的原因，这个注册机构分裂为很多小的注册机构。这也意味着发送查询请求和分析结果也变得麻烦起来，因为要先知道往哪里发送数据，还必须处理多种格式的结果。

不过这还不算什么，Perl模块的作者直到此时还能很好地跟进。因为新的机构产生的速度还不算太快，所以模块作者还能通过发布新模块来跟上。有的人还开发出了能动态扩展服务器地址和格式的模块，比如Vipul Ved Prakash的模块`Net::XWhois`。

随着注册机构的数量逐渐增多，频率逐渐加快，这个办法越来越难适应了。比如`Net::Whois`，这个Dana Hudes维护的模块自从1999年之后就没有更新过，所以也几乎没法用。而`.org`这个顶级域的注册机构调整导致`Net::XWhois`的查找对于这些站点失效。渐渐地，所有模块都无法信赖了。

在我们为此愤愤不平之前，还有个好消息要分享。CenterGate Research Group LLC公司的好心人架设了一个`whois-servers.net`域。在这个域中，他们把所有互联网顶级域的CNAME都注册了。而这些CNAME指向了相应顶级域的注册机构。比如要找到`.com`顶级域的注册机构，可以这样做：

```
$ host com.whois-servers.net
com.whois-servers.net is an alias for whois.verisign-grs.com.
whois.verisign-grs.com has address 199.7.52.74
```

其实使用`Net::DNS`这样的模块来获取这个信息也很容易，不过一个`Net::Whois::Raw`模块已经帮我们完成了这个任务。这个模块的作者是Walery Studennikov，他使用了`whois-servers.net`，并且一直在更新此模块。使用这个模块和使用`Net::Finger`一样容易：

```
use Net::Whois::Raw;

my $whois = whois('example.org');
```

虽然以上代码看上去很直白，不过还是有些细节需要解释。首先，在默认情况下（也就是在我们之前的代码中），这个模块只会为不在模块的硬编码的注册机构表中的顶级域而实际查询`whois-servers.net`名称服务器。要永远依赖`whois-servers.net`以获得顶级域注册机构信息，必须导入并设置一个特殊选项：

```
use Net::Whois::Raw;

$Net::Whois::Raw::USE_CNAMES = 1;
my $whois = whois('example.org');
```

另外还有一个`$OMIT_MSG`选项，设置的方法类似于`$USE_CNAMES`选项。一旦启用这个选项，它会尽可能从结果中去掉WHOIS服务器的版权声明信息（如今的服务器往往会返回这个信息）。不过要小心，这个选项是依赖于作者硬编码的一系列正则表达式来工作的。

除了`$OMIT_MSG`以外，`Net::Whois::Raw`模块并没有对结果做其他分析，它只是简单返回结果。相比之下其他模块都会做一些分析提取的工作，比如`Net::Whois`和

`Net::Xwhois`。应该说，作者的这个决定是明智的，因为注册机构的响应格式往往不太相同，虽然响应中可能会有某些固定的字段（比如Name、Address以及Domain），但没人能保证字段的格式都一致，顺序保持不变。这使得WHOIS信息的分析变得很麻烦，解决这个问题的方法是使用更加复杂的目录服务，比如LDAP。

注意：在结束本节之前还有一个方法要介绍，但是这个方法也有些不确定因素。其实还有几个公共服务器可以作为WHOIS代理服务器，而它们也能接受标准WHOIS查询。在你访问这些代理服务器的时候，它们会帮你查询正确的服务器并且完成格式转换。有两个著名的代理，分别是CenterGate Research Group赞助的whois.geektools.com（参考<http://www.geektools.com>）用于通用WHOIS查询，以及从全局路由表采集信息的whois.pwhois.org（参考<http://pwhois.org>）。两者的使用方法相同，只要用标准whois客户端就可以了，比如`whois -h whois.geektools.com`或者`whois -h pwhois.org 18.0.0.0`。使用这种方法的问题在于：首先它们往往有用量限制（以防滥用）；其次它们依赖于第三方服务器，可能会有服务中断。不过总体来说，偶尔使用它们还是可以的。

LDAP：一种复杂的目录服务

轻量级目录访问协议（Lightweight Directory Access Protocol）或者简称LDAP（也包括其Active Directory实现）是比之前介绍的服务更加强大也更复杂的目录服务。目前有两种版本的LDAP协议被广泛使用，版本2和版本3。下面的介绍会在必要时明确指出具体的版本号。

这个协议是目录访问的行业标准。系统管理员之所以对它趋之若鹜是因为这个标准提供了整合企业信息的能力。除了标准的“公司目录”范例，还能找到一些其他的应用案例：

- NIS到LDAP的网关
- 各种验证数据库（例如，Web上用的）
- 资源的广播（即哪台机器及哪些外设可用）

LDAP还能支持一些更为复杂的目录服务，比如微软的Active Directory，这在稍后的“活动目录服务接口（ADSI）”一节会作介绍。

哪怕你所接触的LDAP只是一个纯粹的电话本，它也值得你去学习一下。LDAP服务器可以使用同样的协议来管理，类似于SQL数据库都可以用SQL来维护一样。Perl在这个协议的管理方面提供了完善的支持，不过我们先得了解LDAP本身才能进行管理。

附录C里有一个LDAP快速入门教程。系统管理员在学习LDAP时最容易遇到的问题就是理解LDAP从它的父协议X.500目录服务那里继承的名词。LDAP虽然是X.500的简化版，不过并没有同时简化那些奇怪的词汇。花点时间阅读附录C会有助于你快速理解那些术语，从而掌握使用Perl管理LDAP的能力。

使用Perl进行LDAP编程

类似于其他Perl系统管理任务，要进行LDAP编程，第一步应该是找合适的模块。LDAP不算是这本书中最复杂的协议，但它并非纯文本协议，这就导致了与LDAP协议进行交互比较麻烦。好在有两个相关的模块可用：`Net::LDAPapi`（也就是`PerlLDAP`和`Mozilla::LDAP`），作者是Leif Hedstrom和Clayton Donley；另外还有Graham Barr的`Net::LDAP`。在本书的第一版，我们展示了这两个模块的代码，不过后来只有`Net::LDAP`保持了更新^[注5]，而`PerlLDAP`则有10年没有改进。尽管你可能还会看到有人在使用`PerlLDAP`开发，不过目前我只能推荐使用`Net::LDAP`，而且书里的代码也只会采用它^[注6]。

作为演示服务器，我们会使用商业化的JES目录服务器（它的前身是Sun One，更早叫做iPlanet，最早叫做Netscape，参考<http://www.sun.com>），以及自由软件OpenLDAP服务器（请参考<http://www.openldap.org>）。这两个服务器支持几乎相同的命令行工具，可以用来验证Perl代码是否如期工作。

建立LDAP连接

连接并验证是任何LDAP客户机/服务器事务中的第一步。对LDAP来说这一步叫做“与服务服务器绑定”。与服务服务器绑定是LDAPv2的必要步骤，不过LDAPv3的要求并没有那么严格。

在绑定到LDAP服务器之后，你就会在某个*distinguished name*（标识名，DN）的上下文中工作，这被称为这个会话的*bind DN*（绑定DN）。这类似于在多用户系统中的登录，你使用的登录账户决定了你对数据的访问权限；对LDAP来说，这个*bind DN*上下文决定了你在LDAP服务器上能访问（包括修改）数据的范围。还有一个特殊的DN，称为*root distinguished name*（这个词没有缩写，为了避免与“relative distinguished name”冲突）。*root distinguished name*是能控制整棵树的DN上下文，这类似于在Unix/Mac OS X上使用*root*登录，或者在Windows上以*Administrator*登录。某些服务器也把这个称为*manager DN*（管理员DN）。

注5： Quanah Gibson-Mount最近开始维护`Net::LDAPapi`，并在2008年1月发布了一个新版本，这是1998年以来的最新版本。

注6： Donley作为这个模块的原始作者，在他的《*LDAP Programming, Management and Integration*》(Manning出版)一书中也使用了`Net::LDAP`。

如果客户机在绑定的时候并没有提供验证信息（比如DN和密码），或者在发送命令之前并没有绑定，那么这就称为匿名绑定（anonymous binding）。匿名绑定的客户机通常只能得到非常有限的数据访问权限。

在LDAPv3规范中有两种绑定方式：简单绑定和SASL绑定。简单绑定使用明文密码来验证。而SASL（Simple Authentication and Security Layer，简单验证和安全层）则支持RFC 2222定义的可扩展验证框架。这个框架能支持在客户机/服务器之间集成的各种验证方案（scheme），比如Kerberos和一次性密码。在客户机连接到服务器时，它会要求使用某种验证机制，如果服务器支持，则会展开这种机制特殊的质询/响应对话。在对话过程中，客户机和服务器之间也可以协商以后的会话所采用的加密方式（比如采用TLS来传输数据）。

有些LDAP服务器和客户机还加入了某种或多种验证方法到简单验证（或者SASL验证）中。这种方法其实是在通过安全套接层（Secure Sockets Layer，SSL）或者它的继任者传输层安全（Transport Layer Security，TLS）的加密通道上运行LDAP。为设置这个通道，LDAP客户机和服务器之间需要完成基于公钥的验证，类似于Web服务器和浏览器为HTTPS所做的。一旦完成了加密通道设置，某些LDAP服务器可以被告知不必再进一步要求客户机提供验证信息。

处理SSL/TLS连接时有两种方式。在LDAPv2的年代，某些服务器会在一个特殊的端口（编号636）监听连接请求。而客户机必须在这个端口登录，在执行任何LDAP指令之前先完成验证。这个实现往往被称为LDAPS，类似于HTTP/HTTPS。然而HTTPS还是有一点与LDAPS不同：LDAPS不是LDAP规范的一部分，所以并非一个“真的”协议，尽管有些服务器支持它。

RFC 2830为此定义了一个LDAPv3协议的扩展。在LDAPv3中，客户机可以从标准LDAP端口（编号389）登录，并且通过Start TLS启动加密的连接。支持这个协议的服务器收到此请求后会开始与客户机展开TLS加密的会话，在会话中可以进行验证以及执行其他LDAP请求。

为了让例子不太复杂，我们只进行简单的验证，也不对会话进行加密。个别特殊的例子除外。

下面就是进行简单绑定与解除绑定的Perl代码：

```
use Net::LDAP;

# 创建一个 Net::LDAP 对象并连接到服务器
my $c = Net::LDAP->new($server, port => $port) or
    die "Unable to connect to $server: $@\n";

# 如果调用 bind() 方法时不提供任何参数，就以匿名方式绑定
# 这里的 $binddn 假设置为类似这样：
# "uid=ucky,ou=people,dc=example,dc=edu"
```

```
my $mesg = $c->bind($binddn, password => $passwd);
if ($mesg->code){
    die 'Unable to bind: ' . $mesg->error . "\n";
}
...
$c->unbind(); # 严格说来, 这条命令并非必需, 不过最好还是写上
```

所有的Net::LDAP方法（比如bind()）都会返回一个消息响答对象。在我们调用此对象的code()方法的时候, 就能获得上次操作的结果代码。成功的操作会返回LDAP_SUCCESS, 也就是0, 上面的代码据此做了检测。

给LDAP通信加密

考虑到网络中鱼龙混杂的状况, 我必须告诉你如何对LDAP通信加密（登录时的验证以及登录之后的操作）。

好在有些很简单的实现方法:

首先你要决定服务器支持的加密方法。可选的方案包括（推荐读者按照顺序考虑）:

1. Start TLS
2. LDAPS
3. SASL

你可能会感到很奇怪, 为什么我把SASL列在最后, 所以我们先解释这个。没错, SASL是最灵活的方法, 但它也是需要最多配置的方法。最常见的使用SASL的场合是用Kerberos（通过SASL^[注7]的GSSAPI机制）作为验证源的时候。还有就是在查询不需要加密（比如公司目录），而修改信息需要加密（比如修改自己的数据）的时候。在这样的情况下我们可以使用SASL, 因为简单绑定是明文的。极个别时候也会在其他情况下使用SASL。

最常见的是前两种协议: Start TLS和LDAPS。这两者对Net::LDAP来说都很容易:

- 对于Start TLS来说, 只需要在使用new()之后但在调用bind()之前调用start_tls()就可以了。
- 对于LDAPS来说, 既可以使用Net::LDAPS模块并给new()方法添加认证相关的参数, 也可以通过给new()提供ldaps:// URI并加上认证相关的参数来使用Net::LDAP模块。

注7: 对于通用的Kerberos验证来说, Graham Barr开发的Authen::SASL软件包（包括它依赖的模块）是可靠的选择。但如果你需要用Kerberos验证方式登录活动目录服务器, 你可以考虑使用Mark Adamson对Cyrus-SASL库的修补（Authen::SASL::Cyrus）。这个模块有些问题, 所以在开始之前请先搜索Net::LDAP模块的邮件列表归档, 以便了解潜在问题。

进行LDAP搜索

LDAP中的字母D代表的是目录（Directory），而针对目录做的最常见操作是搜索。让我们从搜索开始介绍LDAP的功能。LDAP搜索通常如此指定：

在哪里搜索

这被称为`base DN`（基DN）或者`search base`（搜索库）。基DN就是要开始搜索的目录树中的条目的DN。

搜索什么位置

这被称为搜索范围（scope），范围既可以是`base`（只搜索基DN）、`one`（搜索基DN下面一层的条目，不含基DN本身），也可以是`sub`（搜索基DN和它下面的所有条目）。

搜索的对象

这被称为搜索过滤器（search filter）。我们稍后讨论搜索过滤器（search filter）及其指定方式。

返回什么

为了加速搜索操作，你可以设置搜索过滤器为每个找到的条目返回的属性。也可以要求查询仅仅返回属性名，而不含属性值。这对某些情况有用，比如你只关心哪些条目有一个属性，而不关心具体属性值是什么。

小心属性值的引用

在开始Perl编程之前的小提醒：如果你的相对标识名（RDN）中属性的值中带有“+”、“（空格）”、“,”、“'”、“>”、“<”或者“;”等字符，那么请使用引号或者反斜线（\）转义。如果值中含有引号，则只能使用反斜线转义。反斜线自己也需要用更多的反斜线来转义。Net::LDAP::Util 0.32以后的版本提供了`escape_dn_value()`来帮用户完成这样的处理。

没有充分重视引用会导致很多麻烦。当然，如果你能避免在目录的RDN中使用这样的字符就没问题了。

在Perl中进行搜索的代码如下^[注8]：

```
...
my $searchobj = $c->search(base => $basedn,
                           scope => $scope,
                           filter => $filter);
die 'Bad search: ' . $searchobj->error() if $searchobj->code();
```

注8： 因为我们每次都会执行模块的加载、连接对象的创建以及绑定这几个步骤，所以以后这些代码都被表示成省略号。

我们不着急展示完整的代码，现在可以先介绍神秘的\$filter参数。简单的搜索过滤器可以采用如下格式^[注9]：

`<attribute name> <comparison operator> <attribute value>`

这里的<comparison operator>是在RFC 2254中定义的，表9-1列出了更全的操作符。

表9-1：LDAP比较操作符

操作符	含义
=	准确值匹配。如果在<attribute value>中含有*，也可以成为局部匹配（比如cn=Tim O*）
=*	匹配所有含有<attribute name>的值的条目，不论值的内容是什么。通过*（而不是<attribute value>），我们测试条目中是否有某个属性（比如cn=*能匹配那些带有cn属性的条目）
~=	相似值匹配
>=	大于或者等于某值
<=	小于或者等于某值

在你开始为这些符号类似于Perl而开始激动之前，我必须要提醒你，它们和Perl操作符没有任何关系。有两个符号会引起Perl程序员的误会，它们是~=和=*. 第一个符号与正则表达式没有任何关系，它会寻找和某值“接近”的值。这里的“接近”对不同的服务器有不同的含义。大多数服务器使用一种叫做soundex的算法，这个算法开始是为了统计而发明的。它搜索的是那些“听上去”相似的单词，但这两个词很有可能有不同的拼写^[注10]。

另一个可能引起Perl程序员困惑的操作符是=。除了等值比较（字符串和数值）之外，这个符号还可以与前置或后置星号连用，成为通配符，类似于shell对星号的glob处理。例如，cn=fi*会匹配所有common name以“fi”开头的条目；而cn=*ink*则会匹配所有common name中含有“ink”的条目。

我们可以把两个以上的<attribute name> <comparison operator> <attribute value> 搜索过滤器用布尔操作符连接起来，组成一个更加复杂的过滤器。如下所示：

`(<boolean operator> (<simple1>) (<simple2>) (<simple3>) ...)`

注9： 过滤器也有些特殊字符是不能直接使用的。Net::LDAP::Util 0.32版本之后可以用escape_filter_value()来处理这个问题。

注10： 如果你想测试soundex算法，可以考虑使用Mark Mielke的Text::Soundex模块。

有LISP经验的程序员应该不难理解这种语法，其实需要适应的就是组合操作符是在最前面。如果要求条目匹配条件A和B，你可以这样写：`(&(A)(B))`。如要求条目匹配条件A或B或C，那么可以这样写：`(|(A)(B)(C))`。感叹号用来反转某个条件，所以A和非B可以这样写：`(&(A)(!(B)))`。组合后的过滤器还可以进一步与其他过滤器组合产生更加复杂的过滤器。下面的例子可以查询所有在Boston工作的Finkelsteins：

```
(&(sn=Finkelstein)(l=Boston))
```

要查找姓Finkelstein或者Hodgkin的所有人：

```
(|(sn=Finkelstein)(sn=Hodgkin))
```

要查找所有不在Boston工作的Finkelstein：

```
(&(sn=Finkelstein)(!(l=Boston)))
```

要查找所有不在Boston工作的Finkelstein或者Hodgkin：

```
(&(|(sn=Finkelstein)(sn=Hodgkin))(!(l=Boston)))
```

下面的代码能根据LDAP服务器名和LDAP过滤器来搜索，并返回结果：

```
use Net::LDAP;
use Net::LDAP::LDIF;

my $server = $ARGV[0];
my $port = getservbyname('ldap','tcp') || '389';
my $basedn = 'c=US';
my $scope = 'sub';

# 匿名绑定
my $c = Net::LDAP->new($server, port=>$port) or
    die "Unable to connect to $server: $@\n";
my $mesg = $c->bind();
if ($mesg->code){
    die "Unable to bind: " . $mesg->error . "\n";
}

my $searchobj = $c->search(base => $basedn,
                           scope => $scope,
                           filter => $ARGV[1]);
die "Bad search: " . $searchobj->error() if $searchobj->code();

# 打印通过 search() 返回的 $searchobj 中包含的条目
if ($searchobj){
    my $ldif = Net::LDAP::LDIF->new('-', 'w');
    $ldif->write_entry($searchobj->entries());
    $ldif->done();
}
```

下面是节选的样本输出：

```
$ ldapsrch ldap.example.org '(sn=Pooh)'
...
dn: cn="bear pooh",mail=poohbear219@hotmail.com,c=US,o=hotmail.com
mail: poohbear219@hotmail.com
cn: bear pooh
o: hotmail.com
givenname: bear
surname: pooh
...
```

在我们进一步扩展此代码之前，先看看处理`search()`返回的搜索结果的代码。你可能会奇怪的是`Net::LDAP::LDIF`相关的代码在做什么。这其实是一种名叫LDAP Data Interchange Format（LDAP数据交换格式）的数据格式，简称LDIF。稍后几节我们会进一步介绍LDIF。

我们目前更感兴趣的是关于`$searchobj->entries()`调用。这里`Net::LDAP`遵从了RFC 2251定义的编程模型。LDAP搜索结果被返回在`LDAP Message`对象中。所以这段代码调用`entries()`方法来返回这些数据对象中所有条目的列表。接着我们使用相关的`Net::LDAP::LDIF`模块的方法来显示这些条目。

让我们稍微调整一下上面的例子。前面我提到过，可以通过限制搜索返回的属性来加速搜索过程。对于`Net::LDAP`模块来说，这个限制非常简单，只需要给`search()`方法调用加一个额外的参数就可以了：

```
...
# 也可以添加 "typesonly => 1" 来表示仅返回属性类型
# （即需要属性值）
my @attr = qw( sn cn );
my $searchobj = $c->search(base => $basedn,
                           scope => $scope,
                           filter => $ARGV[1],
                           attrs => \@attr);
```

请注意这里`Net::LDAP`用数组引用的方式接受这个新的参数，而不是用数组本身的值的方式来接受。

条目在Perl里的表示

刚才展示的代码可能引起一些关于条目的表示和处理的问题，比如条目本身是如何以Perl的形式存储的以及如何处理它们？作为对LDAP搜索讨论的回应，我们在此会解答部分问题，不过会把更加深入的解答（关于如何添加和修改条目）留到后面的章节去展示。

在使用`Net::LDAP`进行搜索之后，所有的搜索结果都封装在一个`Net::LDAP::Search`对象中返回。要获得此对象中的每个条目的属性，你可以通过以下两种方式之一来实现。

第一种方式，你可以要求此模块帮你把所有以`Net::LDAP::Entry`对象表示的返回条目都转化成一个大型的用户可访问的数据结构。`$searchobj->as_struct()`就是用来完成这个转化的，它的结果是一个“哈希的哈希的列表”数据结构。也就是说，它返回的是一个哈希引用，这个哈希的键是返回条目的DN。这个键对应的值是一个匿名哈希引用，此哈希以属性名为键。而属性名对应的值则是一个匿名数组引用，数组中含有此属性对应的值。请参考图9-1。

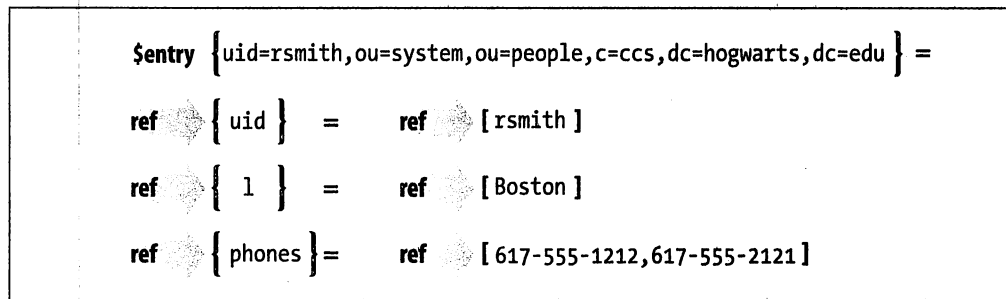


图9-1: `as_struct()`返回的数据结构

要打印出搜索结果数据结构中每个条目的`cn`属性的第一个值，可以使用如下代码：

```

my $searchstruct = $searchobj->as_struct;
foreach my $dn (keys %$searchstruct){
    print $searchstruct->{$dn}{cn}[0],"\n";
}
  
```

另一种方式，你还可以使用以下方法之一来从搜索结果中获取单个条目对象：

```

# 根据指定条目编号返回
my $entry = $searchobj->entry($entrynum);

# 像 Perl 的 shift() 函数那样操作条目列表
my $entry = $searchobj->shift_entry;

# 像 Perl 的 pop() 函数那样操作条目列表
my $entry = $searchobj->pop_entry;

# 将所有条目以列表返回
my @entries = $searchobj->entries;
  
```

一旦获取了单个条目对象，你就可以调用表9-2中列出的方法之一来获取进一步的信息。

表9-2：主要的Net::LDAP条目方法（参考Net::LDAP::Entry说明文档了解更多）

方法名	返回值
<code>\$entry->get_value(\$attrname)</code>	该条目中某属性对应的值。在列表上下文中返回所有值，在标量上下文中只返回第一个值
<code>\$entry->attributes()</code>	该条目中所有属性名的列表

我们可以用像下面这样的写法串联这些方法调用，以获取第一个条目的cn属性的第一个值：

```
my $value = $searchobj->entry(1)->get_value('cn')
```

现在你知道了如何访问搜索结果中的某个属性和值，那么让我们看看如何把这些信息存入目录服务器。

通过LDIF来新增条目

在尝试使用更加通用的方法来给LDAP目录新增条目之前，我们先看看对系统管理员来说最方便的技术。这个技术使用一种叫做LDIF的数据格式，它能够帮你批量载入数据至目录服务器。

LDIF是Gordon Good在RFC 2849中定义的，它为目录条目的文本格式表示提供了标准。下面的LDIF例子来自那个RFC：

```
version: 1
dn: cn=Barbara Jensen, ou=Product Development, dc=airius, dc=com
objectclass: top
objectclass: person
objectclass: organizationalPerson
cn: Barbara Jensen
cn: Barbara J Jensen
cn: Babs Jensen
sn: Jensen
uid: bjensen
telephonenumber: +1 408 555 1212
description: A big sailing fan.

dn: cn=Bjorn Jensen, ou=Accounting, dc=airius, dc=com
objectclass: top
objectclass: person
objectclass: organizationalPerson
cn: Bjorn Jensen
sn: Jensen
telephonenumber: +1 408 555 1212
```

这种格式几乎是一看就明白。在LDIF版本号之后就是每个条目的DN、条目的objectClass定义以及属性的列表。条目之间用空行来分隔。

我们的第一个任务是学写如何从现存的目录条目中写LDIF文件，这样在下一节我们可以读入LDIF。这个功能非常有用，因为这样我们可以用Perl的常用文本操纵惯用语来处理LDIF文件。LDIF有一些特殊的结构（比如对特殊字符和超长行的处理上），所以使用Net::LDAP::LDIF来完成它的生成和读取分析是更好的办法。

在LDAP搜索的例子中，的常用文本操纵惯用已经使用了LDIF格式打印条目。我们把那段代码改成写到文件，而不是直接输出。把下面这行：

```
my $ldif = Net::LDAP::LDIF->new('-', 'w');
```

改成这样：

```
my $ldif = Net::LDAP::LDIF->new($filename, 'w');
```

这样就可以把打印标准输出改成输出到某个文件。

那么现在我们来完成反向的工作，读取LDIF文件而不是生成它们。下面展示的模块对象方法能让我们很容易地把条目加入目录^[注11]。

使用Perl来读取LDIF数据的过程和前面写LDIF文件的例子正好相反。我们会从数据文件中逐个读取条目并且把它转化为条目对象实例，然后使用目录修改方法来处理。Net::LDAP能帮你完成数据读取和分析的过程，所以其实这个任务并不复杂。

警告： 在下面的例子中，我们使用的是`root` DN用户（或者`manager` DN用户）来展示如何完成任务。一般来说应该避免在日常任务中使用此账户。建议设立高权限账户或者高权限用户组来代替`root` DN完成目录管理任务。你可以在自己的环境中尝试应用这个安全策略。

有了Net::LDAP，新增LDIF条目的代码很容易编写：

```
use Net::LDAP;
use Net::LDAP::LDIF;

my $server = $ARGV[0];
my $LDIFfile = $ARGV[1];
my $port = getservbyname('ldap','tcp') || '389';
my $rootdn = 'cn=Manager, ou=Systems, dc=ccis, dc=hogwarts, dc=edu';
my $pw = 'secret';

# 读取在命令行上指定为第二个参数的 LDIF 文件；
# 这里的“r”表示以只读方式打开该文件，如果用“w”则表示可写入
my $ldif = Net::LDAP::LDIF->new($LDIFfile, 'r');
```

注11：LDIF文件还能嵌入`changetype:`指令来指示LDIF读取工具删除或者修改条目信息，而不仅仅是添加。Net::LDAP的Net::LDAP::LDIF::read_entry()方法也能支持`changetype:`。

```

# 取自 Net::LDAP::LDIF 中已经废除的 read() 函数的实现
my ($entry,@entries);
push(@entries,$entry) while $entry = $ldif->read_entry;

my $c = Net::LDAP-> new($server, port => $port) or
    die "Unable to connect to $server: $@\n";

my $mesg = $c->bind(dn => $rootdn, password => $pw);
if ($mesg->code){
    die 'Unable to bind: ' . $mesg->error . "\n"; }

for (@entries){
    my $res = $c->add($_);
    warn 'Error in add for ' . $_->dn().': ' . $res->error()."\n"
        if $res->code();
}

$c->unbind();

```

使用标准LDAP操作来新增条目

现在让我们来看看如何自主完成条目新增的过程，以便跳过读入文件，直接手动创建新条目。Net::LDAP支持两种在目录中创建条目的方法，你可以选择最适合自己的那种。

如果你喜欢用Perl数据结构，也喜欢精简的编程风格，则可以给add()方法提供一个无修饰的数据结构，从而一次性完成创建过程：

```

my $res = $c->add(
    dn => 'uid=jay, ou=systems, ou=people, dc=ccis, dc=hogwarts, dc=edu',
    attr => [ 'cn'          => 'Jay Sekora',
              'sn'          => 'Sekora',
              'mail'        => 'jayguy@ccis.hogwarts.edu',
              'title'       => ['Sysadmin','Part-time Lecturer'],
              'uid'         => 'jayguy',
              'objectClass' => [qw(top person organizationalPerson inetOrgPerson)]
    );
die 'Error in add: ' . $res->error()."\n" if $res->code();

```

这里我们把两个参数传给add()：第一个参数是条目的DN，而第二个参数则是一个匿名数组的引用，数组中存放了属性名-值对的列表。你可能还会注意到title属性的值是用嵌套的匿名数组引用来表达的。

如果你希望循规蹈矩地完成任务，那么可以先创建一个新的Net::LDAP::Entry对象，然后使用add()方法来追加那个对象：

```

use Net::LDAP;
use Net::LDAP::Entry;
...
my $entry = Net::LDAP::Entry->new;

```

```
$entry->dn(
    'uid=jayguy, ou=systems, ou=people, dc=ccs, dc=hogwarts, dc=edu');

# 以下这些 add 语句可以并入一个add()方法中执行
$entry->add('cn' => 'Jay Sekora');
$entry->add('sn' => 'Sekora');
$entry->add('mail' => 'jayguy@ccis.hogwarts.edu');
$entry->add('title' => ['Sysadmin', 'Part-time Lecturer']);
$entry->add('uid' => 'jayguy');
$entry->add('objectClass' =>
    [qw(top person organizationalPerson inetOrgPerson)]);

# 如果喜欢, 也可以调用 $entry->update($c),
# 效果和 add() 一样
my $res = $c->add($entry);
die 'Error in add: ' . $res->error()."\n" if $res->code();
```

这个例子中可能有让人困惑的地方, 那就是关于两个同名add()的使用。其实它们是两个不同的方法, 不过恰巧被赋予了相同的名字。其中一个方法是Net::LDAP::Entry对象的(\$entry->add()), 用来给条目加入新的属性和值。而另一个方法是Net::LDAP连接对象的(\$c->add(\$entry)), 用来把新创建的Net::LDAP::Entry对象加入目录。所以只要你能区分方法名之前的对象, 应该就不会感到困惑了。如果你实在不喜欢这样的重名, 那么可以把第二个add()调用替换成Net::LDAP::Entry update()调用, 这在代码注释里面也提到了。

删除条目

删除目录条目非常简单而且删除后不可恢复, 所以请小心使用。请参考下面的代码:

```
...
my $res = $c->delete($dn);
die 'Error in delete: ' . $res->error(). "\n" if $res->code();
```

请注意delete()操作一次只能删除一个条目。对大多数服务器来说, 想要删除整个子目录树意味着先得使用sub或者one范围来查找出子目录中所有的条目, 然后逐个完成删除。而且只有等到删除了其中所有的条目之后, 才能删除子树的根条目。不过, 接下来的补充内容提供了完成此任务的一些快捷方式。

删除整棵目录子树

到目前为止, 前面描述的方法是唯一正统的删除整棵目录子树的方法, 虽然它非常耗费心力。不过必须指出的是, 还有一些值得尝试的快捷方法:

- 使用别人的代码: OpenLDAP带有一个命令行工具叫做ldapdelete, 它的-r选

项能完成递归删除的功能。虽然这可能不是Net::LDAP爱好者的选择，但是这样真的非常容易完成任务。

- 使用一个非标准的LDAP控制——我们在这一章还没有介绍过LDAP控制（不过稍后会介绍），所以权且把它当作一种魔力删除“药水”好了。

```
my $res =  
    $ldap->delete($dn, control =>  
        {type => LDAP_CONTROL_TREE_DELETE});
```

这段代码有两个难以预测的地方。首先它使用了一个特殊的LDAP控制，而它从来没有进入RFC标准，只是停留在草案阶段（最新的版本是*draft-armijo-ldap-treedelelete-02.txt*）。其次大多数LDAP服务器也不支持它，除了活动目录以外（否则我们就不会介绍它了）。所以请特别留意你使用的服务器类型。

修改条目名

作为最后介绍的LDAP操作，让我们看看如何修改LDAP条目。首先介绍的是如何修改条目的DN或者RDN。

下面展示的代码能调用Net::LDAP模块来修改条目的相对标识名：

```
# $oldDN可以是这样的内容：  
# "uid=johnny,ou=people,dc=example,dc=edu"  
# 而 $newRDN 可以是这样的内容：  
# "uid=pedro"  
my $res = $c->moddn($oldDN,  
    newrdn      => $newRDN,  
    deleteoldrdn => 1);  
die "Error in rename: " . $res->error()."\n" if $res->code();
```

这里是一个关于RDN概念的快速回顾。LDAP服务器以树形结构来存储条目信息，但不能像SNMP或者其他的协议那样使用数字来检索条目。比如在LDAP中不能说“帮我取出第四个分支的第三个条目”，唯一的检索方法是使用路径。这意味着LDAP树的每一个分支都必须有某种区分条目的方法，从而形成唯一的检索路径。因为只有路径的每一段都是唯一的，才能保证条目的全路径是唯一的。^[注12]

确保条目在树的每个分支中唯一的是RDN。这就是为什么我们在这里为了这么简单的一

注12：关于这句话有些需要解释的地方，因为在LDAP目录“树”中其实允许使用类似符号链接的别名，另外也可能还有其他的原因导致同一个条目出现在路径的多处。这并不是我们要讨论的问题，不过这里指出来是为了忠实于有计算机科学背景的读者。

行代码要讨论这么久。在修改RDN的时候，其实改动的正是条目在那个分支中的名字。这个操作并不复杂，关键是要知道我们在做什么。

在我们介绍第二种改名机制之前，还有一个细节需要讨论。你可能注意到了代码中设置的`deleteoldrdn`参数有点特殊。在大多数其他可以修改对象名字的场所，我们并不会考虑修改之后老的名字怎么办的问题。比如在修改文件名之后，文件获得了新名字，所以原来的老名字显然就没有存在的必要了。但是对于LDAP来说，你却有额外的选择：

- 你可以使用`deleteoldrdn => 1`来完成改名，这会一笔勾销以前的RDN。这应该是最合乎常理的选择。
- 你还可以用`deleteoldrdn => 0`来完成改名，这会把旧的RDN信息作为额外值保存在条目中。使用这个选项前请三思。

因为这是一个如此罕见的选项，让我们用例子来说明问题。假定我们要修改的条目看起来像这样：

```
dn: uid=gmarx, ou=People, dc=freedonia, dc=com
cn: Julius Henry Marx
sn: Marx
uid: gmarx
```

如果我们执行了如下的代码：

```
my $oldDN = "uid=gmarx, ou=People, dc=freedonia, dc=com";
my $newRDN = "uid=cspaulding";
my $res = $c->moddn($oldDN, newrdn => $newRDN, deleteoldrdn => 1);
```

那么条目会变成这样：

```
dn: uid=cspaulding, ou=People, dc=freedonia, dc=com
cn: Julius Henry Marx
sn: Marx
uid: cspaulding
```

这里没有发生任何奇怪的事情，结果正是我们需要的。如果把代码的最后一行改成这样：

```
my $res = $c->moddn($oldDN, newrdn => $newRDN, deleteoldrdn => 0);
```

执行之后条目会变成这样：

```
dn: uid=cspaulding, ou=People, dc=freedonia, dc=com
cn: Julius Henry Marx
sn: Marx
uid: gmarx
uid: cspaulding
```

这显然不是我们需要的。所以前面就说过，大多数情况下^{注13}我们应该用1作为deleteoldrdn参数的值。

第二种条目改名的机制可以称为“移花接木”。我们可以通过改名把条目移到目录树的另一个位置。LDAP的第三版引入了一个更加方便的改名机制，从而支持条目在整个目录树范围内的移动。Net::LDAP的moddn()方法已经支持这个机制，我们只需要在调用时注意设置newsuperior参数就可以了。现在我们把这个参数加上：

```
# $oldDN 可以是这样的内容：
# "uid=johnny,ou=people,dc=example,dc=edu"
# 而 $newRDN 可以是这样的内容：
# "uid=pedro"
# $parentDN 可以是这样的内容：
# ou=boxdweller, dc=example,dc=edu
$result = $c->moddn($oldDN,
                    newrdn => $newRDN,
                    deleteoldrdn => 1,
                    newsuperior => $parentDN);
die 'Error in rename: ' . $res->error()."\n" if $res->code();
```

这样，原先存储在\$oldDN的条目现在就可以成为\$parentDN中指定的DN树的成员。使用这个方法移动条目比起之前的协议支持的先delete()再add()方法要高效很多，不过并非所有的服务器都支持。另外还有些可能的问题，比如有些服务器可能并不支持修改带有子树的条目的DN。不过如果你能有效设计目录树结构，那么遇到重定位条目的机会毕竟不会很多。

修改条目属性

我们下面要介绍的操作更加常见，主要是修改条目的属性名和属性值。我们用全局搜索替换作为例子来介绍这些操作。现在假设你的公司下属的某个部门要从Pittsburgh移到Los Angeles，下面的代码能替换所有找到的带有Pittsburgh地点的条目：

```
use Net::LDAP;

my $server = $ARGV[0];
my $port   = getservbyname('ldap','tcp') || '389';
my $basedn = 'dc=ccis,dc=hogwarts,dc=edu';
my $scope  = 'sub';
my $rootdn = 'cn=Manager, ou=Systems, dc=ccis, dc=hogwarts, dc=edu';
my $pw     = 'secret';

my $c = Net::LDAP->new($server, port => $port) or
```

注13：考虑到这个保留老的RDN的例子可能让你摸不着头脑，让我告诉你可能需要这么做的原因：假设现在有两个公司进行了合并，我们必须修改所有的用户名，但是又必须保留老的用户名一段时间。其实有更好的办法来实现这个合并，这里只是为了解释上面的代码。

```

die "Unable to init for $server: $@\n";
my $mesg = $c->bind(dn => $rootdn, password => $pw);
if ($mesg->code){
    die 'Unable to bind: ' . $mesg->error . "\n";
}

my $searchobj = $c->search(base => $basedn, filter => '(l=Pittsburgh)',
                           scope => $scope, attrs => [''],
                           typesonly => 1);
die 'Error in search: '.$searchobj->error()."\n" if ($searchobj->code());

if ($searchobj){
    @entries = $searchobj->entries;
    for (@entries){

        # we could also use replace {'l' => 'Los Angeles'} here
        $res=$c->modify($_->dn(), # dn() 给出该条目的 DN 信息
                      delete => {'l' => 'Pittsburgh'},
                      add      => {'l' => 'Los Angeles'});

        die 'unable to modify, errorcode #'.$res->error() if $res->code();
    }
}
$c->unbind( );

```

这段代码里面最主要的部分是通过`modify()`方法来实现的。这个方法接受条目的DN作为参数，另外还有一组如何修改条目的参数。表9-3列出了主要的修改参数。

表9-3: Net::LDAP 条目修改参数

参数	效果
<code>add => {\$attrname => \$attrvalue}</code>	给条目增加一个新的属性，包括属性值
<code>add => {\$attrname => [\$attrvalue1, \$attrvalue2...]}</code>	给条目增加一组属性和相应的属性值
<code>delete => {\$attrname => \$attrvalue}</code>	删除条目中的某个属性，包括相应的值
<code>delete => {\$attrname => []}</code>	删除某个或者某组属性以及相应的值
<code>delete => [\$attrname1, \$attrname2...]</code>	
<code>replace => {\$attrname => \$attrvalue}</code>	就像add那样，但是能替换相应的属性值如果 \$attrvalue是一个空匿名列表的引用（[]），那么这个调用的效果和delete相同

请特别注意表9-3中的标点符号。某些参数使用的是匿名哈希引用，有些则使用匿名数组引用。请勿混用，否则肯定会出问题。

如果你发现自己需要对某个条目执行多个修改（如同我们的代码演示的），那么可以

把多个参数合并在一个`modify()`调用中。然而这可能会导致问题，好比你写了如下的`modify()`调用：

```
$c->modify($dn,replace => {'1' => 'Medford'},
           add      => {'1' => 'Boston'},
           add      => {'1' => 'Cambridge'});
```

这里并没有办法保证新增属性会发生在修改属性之后，这就会导致代码的行为结果不可预测。

如果你希望操作按照设定的顺序执行，那么可以使用一个稍微奇怪一些的语法。在使用一组分离的参数以外，其实你还可以传入一个命令数组。在下面的新版本中，`modify()`接受一个名为`changes`的参数，其值为一个列表。列表中含有一组名-值对，名也就是要执行的操作，值就是要执行要操作数据的匿名数组的引用。如果我们需要确保上面例子中的代码按照顺序执行，可以如下编写：

```
$c->modify($dn, changes =>
[ replace => ['1' => 'Medford'],
  add      => ['1' => 'Boston'],
  add      => ['1' => 'Cambridge']
]);
```

请再次注意标点符号：这个例子里面使用的和上面的例子使用的又不相同。

更深入的LDAP主题

如果你到目前为止都还读得津津有味，那么其实你已经掌握了基础的技巧，能够从Perl中使用LDAP的大多数功能了。如果你非常希望看到如何把这些技能都使用起来，完成更加复杂的任务，那么可以跳过下面这一段，直接参考“综合练习”一节，在结束之后再回来。如果你觉得还是愿意按着顺序来学习，那么下面的这些高级主题能让你有更加坚实的基础知识。

推介和引用

在区分LDAP推介（referral）和引用（reference）的时候，我们需要做的只是不要在两者之间建立任何关联。在LDAP第二版的阶段，推介还非常简单（简单到没有规范中提到）。如果你查询LDAP第二版的某个服务器它没有的数据，那么它会返回一个默认的推介，意思是“我没有你需要的数据，请在这个LDAP URL查询”。于是LDAP客户机可以使用那个RFC 2255中定义的URL来确定应该查询的服务器和基DN。例如，某个客户机向服务器发起了对关于`ou=sharks,dc=jeromerobbins,dc=org`的`ou=jets,dc=jeromerobbins,dc=org`的查询，而服务器的回答是：“对不起，请转而查找`ldap:///robertwise.org/ou=jets,dc=robertwise,dc=org`。”那么客户机可以转而连

接robertwise.org站点上运行的LDAP服务器。

LDAP第三版对这个概念进行了拓展，使它变得更加复杂。现在服务器如果收到了不能处理的查询，它会返回一个URL或者一组URL供客户选择。客户可以挑选合适的URL来进一步查询。

在LDAP第三版中对这个概念进行的另一个拓展是继续引用（continuation reference），这其实是一种特殊的推介^[注14]，只发生在LDAP搜索中。如果服务器在处理搜索中发现另外一个服务器中有关于此搜索的更多数据，那么它可以在结果中返回一个继续引用，这个继续引用的意思是：“我不能完全回答问题，不过我知道谁能做。请查询这个（或者这些）URL来得到完整的答案。”客户机此时可以自行决定是否要继续查询其他服务器来完成查询。继续引用一般在处理大型目录树的时候会比较好用，因为这样一来可以把部分子树部署在其他服务器上，从而提高负载平衡度。

让我们看看这些过程如何使用Perl代码来实现。尽管这两者有些关联，我们在例子中还是分开介绍推介和继续引用。处理推介需要这些步骤：

1. 在搜索结束之后，看看是否收到了推介。如果没有，可以继续处理结果。
2. 如果收到了推介，那么把LDAP URL^[注15]从响应中取出来，并且解析此URL以备连接使用。
3. 根据URL绑定到合适的服务器并且对它进行查询。回到第一步，因为可能我们还会收到推介。

以上的步骤很容易转化成代码：

1. 检查推介：

```
use Net::LDAP qw(LDAP_REFERRAL); # 务必导入该常量
use URI::LDAP; # 我们将要用该模块解析 LDAP URL

# 和以往一样进行绑定
...
# 像以往那样搜索
my $searchobj = $c->search(...);

# 看看是否收到了推介
```

注14：为了帮助你记住推介和引用的区别，每次提到引用的时候，我总是说“继续引用”。

注15：RFC 2251（也就是LDAP第三版的规范）其中说到关于多个推介URL的选择：“所有返回的URL应该在查询上面是等效的。”这意味着你可以随意选择使用某一个URL。你可以使用最简单的策略（选择第一个或者随机选择），也可以使用折中的策略（挑选ping值最低的），或者使用最复杂的策略（选择距离你的网络最近的那个），这些都是你的自由。

```
if ($searchobj->code() == LDAP_REFERRAL) {
```

2. 提取LDAP URL:

```
# 上面的返回代码表示现在确实收到了一些推介，先把它们取出来
my @referrals = $searchobj->referrals();
# RFC 2251 规范允许我们随意取一个来用，那就用第一个好了
my $uri = URI->new($referrals[0]);
```

3. 使用URI::LDAP模块从URL中解析出必要的信息，用来再次绑定并查询:

```
$c->unbind();
my $c = Net::LDAP-> new ($uri->host(), port => $uri->port()) or
die 'Unable to init for ' . $uri->$host . ":\n";
my $mesg = $c->bind(dn => $rootdn, password => $pw);
if ($mesg->code){
    die 'Unable to bind: ' . $mesg->error . "\n";
}

# RFC 2251 规范要求我们在搜索推介 URL 时必须使用过滤器;
# 否则, 就应该用原来的过滤器
#
# 注意: 我们将要用的是 $uri->_filter() 而非 $uri->filter()
# 因为后者如果没有在该 URL 中给定过滤器就只返回默认字符串。
# 而我们希望用原来的过滤器, 不是默认的 (objectClass=*)
$searchobj = $c->search(base => $uri->dn(),
                        scope => $scope,
                        filter => $uri->_filter() ? $uri->_filter() :
                                                $filter,
                        ...);
}
```

你会发现，推介处理类似于比较复杂的错误处理，这样会更容易理解。因为你查询的服务器其实是在说：“对不起，无法完成查询。请重试，不过记住使用这个服务器的这个端口以及这个基DN，另外还可以继续使用原来的过滤器。”

请注意上面的代码并非足够的复杂，也并非完全周密。首先因为，RFC 2251中声明的是所有的操作都可能会收到推介，但这段代码只处理了查询的推介（没有处理绑定的推介）。不过我建议你仔细考虑一下，是否有必要为了这个修改你的代码。如果你要把验证信息发到一个并不认识的服务器，那么最起码你得仔细验证推介要绑定的服务器的证书。另外，其他的那些LDAP操作导致的推介也值得仔细审查。

其次，除非是仔细设计过目录的结构，否则没有办法断定推介服务器上的操作不会再次导致推介。这种情况对客户端来说是非常麻烦的，所以在实际应用中很少见到，但确实有这种可能。

最后，还是实际应用中罕见的情况，代码并没有判断循环推介的可能性。也就是说服务器A推介服务器B，而B反过来又推介服务器A。当然可以通过不断更新推介列表来避免这个死循环，但这应该也非常罕见。

现在你已经能妥善处理推介了，下面的主题是继续引用。可以说继续引用并不难处理，它们只会发生在搜索操作中，而且它们只是在搜索成功开始之后才有可能发生（也就是说你要搜索的位置确实在目录树中存在）。和之前介绍的推介不同，收到了继续引用并不意味着出错，所以也不必重新搜索。继续引用有些类似于欠债人和倒霉的债主的关系。如果你的程序是债主，那么它会问服务器有没有足够的信息，而服务器这时候回答：“对不起，我手上的信息不够，不过你可以去这三个位置找更多的信息。”这样你的程序就必须走遍那三个地方来获取全部的信息，而不像推介的情况（只要选择其中一个就行了）。不幸的是，这些地方也可能会给你同样的继续引用。

从编程的角度来说，继续引用和推介的差别主要是：

1. 检查并注册的方法不相同。对于推介来说，你可以检查操作的结果代码并且调用 `referrals()` 方法来注册。对于继续引用，你得从返回结果中检查是否有继续引用，若有则调用 `references()` 方法进行注册。

```
..... # 这里省略绑定和搜索操作
if ($searchobj){
    my @returndata = $searchobj->entries;
    foreach my $entry (@returndata){
        if ($entry->isa('Net::LDAP::Reference')){
            # @references 内包含一系列 LDAP URL
            push(@references,$entry->references());
        }
    }
}
```

2. 与推介不同，我们不能挑选URL，继续引用的URL必须全部访问。大多数人使用递归子例程^[注16]来写这种代码：

```
..... # 假设已经完成搜索，并且得到的是继续引用
foreach my $reference (@references){
    ChaseReference($reference)
}

sub ChaseReference ($reference){
    my $reference = shift;

    # 这段代码应该很熟悉了，基本上我们一直是在借用之前例子中的代码

    # 解析 LDAP URL，绑定相应的服务器，再进行搜索操作
    my $uri = URI->new($reference);
    my $c = Net::LDAP-> new ($uri->host(), port => $uri->port()) or
        die 'Unable to init for ' . $uri->$host . " : $@\n";
    my $mesg = $c->bind(dn => $rootdn, password => $pw);
    if ($mesg->code){
        die 'Unable to bind: ' . $mesg->error . "\n";
    }
}
```

注16：需要帮助回忆递归概念的话，请参考第2章。

```

my $searchobj = $c->search(base => $uri->dn(),
                           scope => $scope,
                           filter => $uri->_filter() ? $uri->_filter() :
                                                           $filter,
                           ...);
# 假设得到结果，则收集各条目信息和引用到不同的列表
if ($searchobj){
    my @returndata = $searchobj->entries;
    my @references = ();
    foreach my $entry (@returndata){
        if ($entry->isa('Net::LDAP::Reference'){
            # @references will contain a list of LDAP URLs
            push(@references,$entry->references());
        }
        else { push @entries, $entry );
    }
}

# 现在，逐个跟随上次搜索结果中找到的引用
# （这将是一个递归过程）
foreach my $reference (@references){
    ChaseReference($reference)
}
}

```

如果你是那种打破沙锅问到底的人，可能会问这些操作是否有可能导致推介，而且如果有推介的话代码是否应该处理它们。我只能说没错，应该处理。现在还有别的问题么？

严格意义上来说，这里的代码都只是为介绍概念而编写的。为了避免推介和继续引用的混淆，这里并没有深入探讨边界情况。如果你希望最大可能地提高代码的健壮度，那么应该对每个LDAP操作都包裹上推介处理子例程，并且对搜索操作都使用该特别处理推介和继续引用的子例程。

控制和扩展

我听到过的最好的LDAP控制的比喻来自于Gerald Carter的著作《LDAP System Administration》（O'Reilly出版）。书中Carter说它们对于LDAP操作来说就好像是助词，它们能润色或提升常规LDAP操作的效果。比如，想要让服务器返回排序之后的结果，那么你可以使用Server Side Sorting控制（在RFC 2891中定义）。让我们假定服务器支持这个特性（并非所有的服务器都支持，Sun JES Directory支持，而OpenLDAP不支持），看看代码该如何编写。

大多数情况下，第一步是找到某个控制对应的Net::LDAP::Control的子模块（每个常见的控制应该都有一个模块）^[注17]。这里使用的是Net::LDAP::Control::Sort，我们用它来创建一个对象：

注17：如果你想尝试的控制功能无法被服务器支持，请别灰心。Net::LDAP里面有很多的控制范例，可以挨个试试看。

```

use Net::LDAP;
use Net::LDAP::Control::Sort;

...

# 创建一个控制对象，我们将用它来对姓氏 (surname) 排序
$control = Net::LDAP::Control::Sort->new(order => 'sn');

```

有了控制对象之后，只需要用它来修改搜索就可以了：

```

# 这会返回排序后的条目
$searchobj= $c->search (base    => $base,
                        scope    => $scope,
                        filter   => $filter,
                        control => [$control]);

```

有些控制使用起来更加复杂，不过这里已经有了基本的概念。

扩展（有时也称为“扩展操作”）类似于控制，只不过功能更加强大。区别在于它并不是润色某个常规的LDAP操作，而是通过扩展LDAP协议来引入更多的新操作。例如可以通过Start TLS (RFC 2830) 来引入数据传输安全机制，还可以通过LDAP Password Modify (RFC 3062) 来引入LDAP服务器密码修改机制。

在Perl中使用扩展非常容易，因为各种常见的扩展模块都已经集成在Net::LDAP中了。比如要使用Password Modify只需要这样：

```

use Net::LDAP;
use Net::LDAP::Extension::SetPassword;

..... # 这里省略连接和绑定操作
$res = $c->set_password( user      => $username,
                        oldpassword => $oldpw,
                        newpassword => $newpw, );
die 'Error in password change : ' . $res->error()."\n" if $res->code();

```

如果你要使用的扩展还没有集成进来，那么最好的办法就是从Net::LDAP::Extension::SetPassword这样的模块拷贝一个文件，并且修改内容来实现需要的功能。

你读到这里的时候可能会有一个问题，那就是：“如果没有服务器文档（或者源代码），如何知道我的服务器支持哪些控制和扩展？”其实这时候你可以查询根DSE，这正是下一节的主题。

根DSE

这个主题最复杂的事情是弄清楚那些从X.500集成过来的臃肿的术语。下面就列出它们本来的含义：

DSE是一个与DSA相关的条目，那么什么是DSA？

DSA是一个目录系统代理。那么什么是目录系统代理？

目录系统代理也就是一个LDAP服务器。

除了让你当众炫耀X.500术语知识以外，这些对我们有什么价值呢？根DSE的价值在于这个条目存储了关于目录本身的信息。根据RFC 2251定义，根DSE中起码有如下属性：

namingContexts

这个服务器支持的目录树，或称为后缀（比如dc=ccis, dc=hogwarts, dc=edu）。

subschemaSubentry

在哪里可以找到LDAP服务器支持的模式（schema）（请参考附录C了解LDAP模式）。

altServer

根据RFC 2251，指定一组在本服务器离线时可以查询的服务器。这个信息可以在第一次与服务器联系的时候记录下来，这样万一服务器崩溃，你还可以找到其他可用的服务器。

supportedExtension

这个服务器支持的扩展列表。

supportedControl

这个服务器支持的控制列表。

supportedSASLMechanisms

这个服务器支持的SASL机制（比如Kerberos）列表。

supportedLDAPVersion

这个服务器使用的LDAP协议版本（目前来说应该是2或3）

获得这些信息非常容易。Net::LDAP的Net::LDAP::RootDSE模块可以这样调用：

```
use Net::LDAP;
use Net::LDAP::RootDSE;

my $server = 'ldap.hogwarts.edu';

my $c = Net::LDAP->new($server) or
    die "Unable to init for $server: $@";

my $dse = $c->root_dse();

# 让我们来看看有哪些后缀可以在该服务器上找到
print join("\n", $dse->get_value('namingContexts')), "\n";
```

这段代码可能返回如下的信息（即服务器支持的后缀列表）：

```
dc=hogwarts,dc=edu
o=NetscapeRoot
```

你可能已经注意到，我们没有一如既往地省略绑定步骤。这是因为`Net::LDAP::RootDSE`会自动完成匿名绑定，然后使用`search()`来搜索我们感兴趣的信息。如果你去观察LDAP服务器日志，应该会发现如下的记录：

```
[16/May/2004:21:25:46 ?0400] conn=144 op=0
msgId=1 - SRCH base="" scope=0 filter="(objectClass=*)" attrs="subsch
emaSubentry namingContexts altServer supportedExtension supportedControl
supportedSASLMechanisms supportedLDAPVersion"
```

这意味着我们使用空的基DN进行搜索（这也就是对根DSE的搜索），搜索范围是base（日志中的“0”），过滤器则是返回全部属性。现在你又学到了一招，以后如果`Net::LDAP::RootDSE`返回的属性不全，你应该可以自己搜索了。

DSML

我们最后一个高级主题是目录服务标记语言（Directory Services Markup Language，DSML）。DSML有两种，第一版和第二版。我们可以把DSML第一版当成是LDIF的XML改进版。从字面上理解，DSML与“通过LDIF来新增条目”一节介绍的LDIF的区别主要是DSML使用XML格式来展现条目数据。说它是一种改进，主要是因为它不仅是条目的表示标准，还能支持目录模式（schema）的表示（参考附录C）。以上是优点，不过它也有些小问题，主要是第一版的DSML在表示目录操作上不如LDIF（比如`changetype:delete`）。这个缺陷在DSML的第二版得到了解决，这也导致它变得更加复杂。到目前为止，Perl还没有很好地跟进第二版，所以模块只能支持第一版的DSML。

然而，如果第一版的DSML就是你的选择，那么`Net::LDAP::DSML`提供了很方便的方法来导出此格式的数据（不过目前为止，这个模块还不能读取此格式的文件^[注18]）。操作步骤和之前写LDIF的相似：

```
use Net::LDAP;
use Net::LDAP::DSML;

open my $OUTPUTFILE, '>', 'output.xml'
or die "Can't open file to write: ${!}\n";

my $dsml = Net::LDAP::DSML->new(output => $OUTPUTFILE,
                                pretty_print => 1)
or die "OUTPUTFILE problem: ${!}\n";
```

注18：如果你需要读入DSML，你可以使用任何一种XML读取模块（比如`XML::Simple`）来读入数据，然后使用在“使用标准LDAP操作来新增条目”一节介绍的`Net::LDAP`调用。

```

..... # 绑定并搜索以取得 @entries

$dsml->start_dsml();

foreach my $entry (@entries){
    $dsml->write_entry($entry);
}

$dsml->end_dsml();
close $OUTPUTFILE;

```

在运行以上代码的时候，你会得到如下的输出（加入了手动缩进，以增强层次感）：

```

<?xml version="1.0" encoding="UTF-8"?>
<dsml:dsml xmlns:dsml="http://www.dsml.org/DSML">
  <dsml:directory-entries>
    <dsml:entry dn="ou=People, dc=hogwarts, dc=edu">
      <dsml:objectclass>
        <dsml:oc-value>top</dsml:oc-value>
        <dsml:oc-value>organizationalunit</dsml:oc-value>
      </dsml:objectclass>
    </dsml:entry>
  </dsml:directory-entries>
  <dsml:directory-entries>
    <dsml:entry dn="uid=colinguy, ou=People, dc=hogwarts, dc=edu">
      <dsml:attr name="cn">
        <dsml:value>Colin Johnson</dsml:value>
      </dsml:attr>
      <dsml:attr name="uid">
        <dsml:value>colinguy</dsml:value>
      </dsml:attr>
      <dsml:objectclass>
        <dsml:oc-value>top</dsml:oc-value>
        <dsml:oc-value>person</dsml:oc-value>
        <dsml:oc-value>organizationalPerson</dsml:oc-value>
        <dsml:oc-value>inetorgperson</dsml:oc-value>
      </dsml:objectclass>
    </dsml:entry>
  </dsml:directory-entries>
</dsml:dsml>

```

有人可能会好奇：“为什么要使用DSML，而不是LDIF呢？”这个问题是很合理的。DSML是为了以XML格式展现目录数据而设立的（在第二版本中还能展示目录操作）。如果你的工作是进行跨机构的目录数据共享，或者找到了其他的用处，那么DSML可能会有帮助。但是如果你只是打算坚守在LDAP的应用上，对XML格式提供的其他数据整合并不感兴趣，那么LDIF会是更好的选择。LDIF不但更加简单，也经过更多测试，而且有更好的跨服务器支持。

综合练习

现在我们已经完全介绍了LDAP的主要领域（连一些次要的也介绍了），让我们开始编写系统管理相关的脚本吧。我们会把之前在第5章准备的主机数据库导入LDAP服务器，并且使用LDAP查询来获得一些有意义的结果。下面是那个文件的节选，请注意其中的文本格式：

```
name: shimmer
address: 192.168.1.11
aliases: shim shimmy shimmydoodles
owner: David Davis
department: software
building: main
room: 909
manufacturer: Sun
model: M4000
--
name: bendir
address: 192.168.1.3
aliases: ben bendoodles
owner: Cindy Coltrane
department: IT
building: west
room: 143
manufacturer: Apple
model: Mac Pro
--
```

我们要做的第一件事是配置目录服务器，使它能接收这些数据。我们使用了非标准属性，所以需要修改服务器的模式，这在不同的服务器上有不同的方法。比如对于Sun JES Directory Server来说，它有一个很友善的目录服务器控制台GUI，能帮我们完成这个任务。而对于其他服务器来说，往往需要修改文本配置文件。对于OpenLDAP来说，我们需要在某个被主配置文件引用的文件中定义自己的主机对象类：

```
objectclass machine
requires
    objectClass,
    cn
allows
    address,
    aliases,
    owner,
    department,
    building,
    room,
    manufacturer,
    model
```

一旦完成了服务器配置，我们就可以考虑导入数据了。一个方式是使用LDIF进行批量导

入。如果你开始觉得我们的样本文件和LDIF看上去有些类似，那么你就对了。这个相似性使得我们的翻译工作得到简化，不过还是有几个小陷阱需要注意：

续行符

我们的平面文件（flat-file）数据库中没有任何条目的值会跨行，如果有的话我们就得注意产生的输出必须遵从LDIF标准。而标准里面明确说了，续行符是行首的单个空格。

条目分隔符

我们的数据库使用了可爱的`--\n`字符序列来分隔条目。而LDIF条目之间必须使用两个换行符号（也就是一个空行）来分隔。所以我们必须在输入中对此分隔符进行替换。

属性分隔符

目前我们的数据只有一个多值属性：`aliases`。LDIF对多值属性的展现方法是每行列出一个值。所以如果遇到多别名的主机条目，就必须使用特殊的代码来为每个别名打印一行。如果不是这个数据格式的差异，我们完全可以使用一行代码完成格式的转换。

不过就算有这么多小陷阱，转换程序还是挺简单的：

```
my $datafile    = 'database';
my $recordsep   = "--\n";
my $suffix      = 'ou=data, ou=systems, dc=ccis, dc=hogwarts, dc=edu';
my $objectclass = <<"EOC";
objectclass: top
objectclass: machine
EOC

open my $DATAFILE, '<', $datafile or die "unable to open $datafile: ${!}\n";

print "version: 1\n"; #

while (<$DATAFILE>) {
    # 打印每个条目的头部
    if (/^name:\s*(.*)/){
        print "dn: cn=$1, $suffix\n";
        print $objectclass;
        print "cn: $1\n";
        next;
    }
    # 处理多值别名属性
    if (/^aliases:\s*(.*)/){
        my @aliases = split;
        foreach my $name (@aliases){
            print "aliases: $name\n";
        }
        next;
    }
}
```

```

# 处理每条记录末尾的分隔符
if ($_eq $recordsep){
    print "\n";
    next;
}
# 否则, 仅仅打印得到的属性
print;
}

close $DATAFILE;

```

如果我们运行这段代码, 它就会输出如下的LDIF文件 (节选):

```

version: 1
dn: cn=shimmer, ou=data, ou=systems, dc=ccis, dc=hogwarts, dc=edu
objectclass: top
objectclass: machine
cn: shimmer
address: 192.168.1.11
aliases: shim
aliases: shimmy
aliases: shimmydoodles
owner: David Davis
department: software
building: main
room: 909
manufacturer: Sun
model: M4000

dn: cn=bendir, ou=data, ou=systems, dc=ccis, dc=hogwarts, dc=edu
objectclass: top
objectclass: machine
cn: bendir
address: 192.168.1.3
aliases: ben
aliases: bendoodles
owner: Cindy Coltrane
department: IT
building: west
room: 143
manufacturer: Apple
model: Mac Pro
...

```

有了这个LDIF文件, 我们可以使用服务器自带的批量加载程序来完成数据载入。比如对于OpenLDAP和Sun JES Directory Server来说, 可以使用*ldif2ldb*来读入LDIF文件, 并且直接转换成服务器的后端数据格式 (甚至不需要通过LDAP)。尽管你只能在服务器关闭期间这样做, 不过在速度上它是最快的。如果你不能关闭服务器, 还可以考虑使用我们之前开发的LDIF读入Perl代码来载入数据。

为了提供更多的选择, 还可以使用下面的代码来跳过LDIF格式直接导入数据到LDAP服务器:

```

use Net::LDAP;
use Net::LDAP::Entry;

my $datafile = 'database';
my $recordsep = '--';
my $server = $ARGV[0];
my $port = getservbyname('ldap','tcp') || '389';
my $suffix = 'ou=data, ou=systems, dc=ccis, dc=hogwarts, dc=edu';
my $rootdn = 'cn=Manager, ou=Systems, dc=ccis, dc=hogwarts, dc=edu';
my $pw = 'secret';

my $c = Net::LDAP->new($server,port => $port) or
    die "Unable to init for $server: $@\n";
my $mesg = $c->bind(dn => $rootdn,password => $pw);
if ($mesg->code){
    die "Unable to bind: " . $mesg->error . "\n";
}

open my $DATAFILE, '<', $datafile or die "unable to open $datafile:$!\n";

while (<$DATAFILE>) {
    chomp;
    my $entry;
    my $dn;
    # 看到新记录的开头，就创建一个新的条目对象实例
    if (/^name:\s*(.*)/){
        $dn="cn=$1, $suffix";
        $entry = Net::LDAP::Entry->new;
        $entry->add('cn',$1);
        next;
    }
    # 对于多值属性的特殊处理
    if (s/^aliases:\s*//){
        $entry->add('aliases',[split()]);
        next;
    }

    # 如果到达记录尾部，则将该记录添加到服务器
    if ($_ eq $recordsep){
        $entry->add('objectclass',['top','machine']);
        $entry->dn($dn);
        my $res = $c->add($entry);
        warn "Error in add for " . $entry->dn() . ':' .
            $res->error()."\n" if $res->code();
        undef $entry;
        next;
    }

    # 添加其他所有属性
    $entry->add(split(':',$_)); # 假设都是单值属性
}

close $DATAFILE;
$c->unbind();

```

现在我们已经把数据导入了服务器，可以开始做些有趣的事情了。为了节约篇幅，后面的例子会省略那些开头的变量设置和绑定服务器的代码。

那么我们如何使用LDAP服务器端的数据呢？我们可以用它来生成host文件：

```
use Net::LDAP;

...

my $searchobj = $c->search (base => $basedn,
                             scope => 'one',
                             filter => '(objectclass=machine)',
                             attrs => ['cn','address','aliases']);
die "Bad search: " . $searchobj->error() if $searchobj->code();

if ($searchobj){
    print "#\n# host file - GENERATED BY $0\n
          # DO NOT EDIT BY HAND!\n#\n";
    foreach my $entry ($searchobj->entries()){
        print $entry->get_value(address),"\t",
              $entry->get_value(cn)," ",
              join(' ', $entry->get_value(aliases)),"\n";
    }
}
$c->close();
```

下面就是输出：

```
#
# host file - GENERATED BY ldap2hosts
# DO NOT EDIT BY HAND!
#
192.168.1.11    shimmer shim shimmy shimmydoodles
192.168.1.3    bendir ben bendoodles
192.168.1.12   sulawesi sula su-lee
192.168.1.55   sander sandy mickey mickeydoo
```

我们还可以搜索所有Apple制造的主机的名称：

```
use Net::LDAP;

...

my $searchobj = $c->search(base => $basedn,
                           filter => '(manufacturer=Apple)',
                           scope => 'one',
                           attrs => ['cn']);
die "Bad search: " . $searchobj->error() if $searchobj->code();

if ($searchobj){
    foreach my $entry ($searchobj->entries){
        print $entry->get_value('cn'),"\n";
    }
}

$c->unbind();
```

下面就是输出：

```
bendir  
sulawesi
```

我们可以生成主机拥有者的列表：

```
use Net::LDAP;  
...  
my $searchobj = $c->search(base => $basedn,  
                           filter => '(manufacturer=Apple)',  
                           scope => 'one',  
                           attrs => ['cn','owner']);  
die 'Bad search: ' . $searchobj->error() if $searchobj->code();  
  
my $entries = $searchobj->as_struct;  
  
foreach my $dn (sort byOwner keys %{entries}){  
    print $entries->{$dn}->{owner}->[0]. ":\t" .  
          $entries->{$dn}->{cn}->[0]."\n";  
}  
  
# 按照拥有者的名字而不是它的DN值排序  
sub byOwner  
{ $entries->{$a}->{owner}->[0] <=> $entries->{$b}->{owner}->[0] }
```

下面就是输出：

```
Alex Rollins:  sander  
Cindy Coltrane: bendir  
David Davis:   shimmer  
Ellen Monk:    sulawesi
```

我们还能查看当前的用户ID是否就是这台Unix机器的拥有者（可以作为某种伪验证机制）：

```
use Net::LDAP;  
use Sys::Hostname;  
  
$user = (getpwuid($<))[6];  
  
my $hostname = hostname;  
my $hostname =~ s/\.*.*//;          # 去掉域名部分，留下主机名  
...  
  
my $searchobj = $c->search (base      => "cn=$hostname,$suffix",  
                           scope     => 'base',  
                           filter    => "(owner=$user)"  
                           typesonly => 1);  
  
if ($searchobj){  
    print "Owner ($user) can log on to machine $hostname.\n";  
}
```

```
else {  
    print "$user is not the owner of this machine ($hostname).\n";  
}
```

这些代码都能让你体会到系统管理员是如何用Perl来进行LDAP访问的，也可能会给你一些开发新程序的灵感。在下面一节我们会把这个体验提升到另外一个高度，让你看看LDAP系统是如何支撑起整个管理框架的。

并非（真正的）数据库

在我们进一步讨论ADSI之前，我希望能有机会指出避免使用LDAP的场合。你可能会听到有个声音说，为什么不把LDAP作为所有信息的中心仓库，就像第7章中讨论的那样呢？毕竟微软也在活动目录中这么做了呀？

这是一个值得探讨的问题，起码我是这么认为的，LDAP并非那些自主开发系统的最佳选择。LDAP虽然给了我们数据库的感觉，但它并没有关系数据库的功能。它对于数据内容的约束非常少，并不使用关系模型构造，也不支持复杂的查询语言。

我的建议是尽量使用关系数据库来存储大多数信息，并且从数据库导入信息到LDAP服务器。这会给你两种模型的最佳搭配，避免在LDAP不能胜任的场合滥用它。微软确实能付出这个代价开发足够的代码，从而使得管理工具都支持LDAP作为中央数据库。但对你来说这么做的成本可能过于庞大。要是你决定走这条路，请先仔细权衡。

活动目录服务接口（ADSI）

在这一章剩余的部分，我们会讨论一种依赖于平台的目录服务，这种服务的体系结构依赖于刚刚介绍过的LDAP服务。

微软在LDAP基础上开发了一种复杂的目录服务，名叫活动目录（Active Directory），集成在Windows的管理框架中。活动目录成为了Windows网络中重要配置信息（如用户、组、系统策略和软件安装信息等）的仓库。

在开发活动目录的过程中，微软的技术人员意识到需要某种高层编程接口。他们把开发出来的接口称为活动目录服务接口（Active Directory Service Interface, ADSI）。更棒的是，微软的开发人员意识到这个新的框架可以被扩展到其他系统管理领域，比如用来管理打印机和系统服务。这个决定对那些使用脚本来自动完成系统管理任务的人来说非常重要。我们先介绍一些基本的概念和词汇，然后再展示如何在实践中使用这些功能。

ADSI基础

你可以把ADSI理解成任何需要加入ADSI平台的目录服务的外包装。LDAP、SAM (Security Accounts Manager,安全账户管理器,即本地/WinNT域风格)数据库或者Novell目录服务都可以成为ADSI的供应者(这是ADSI集成方面的术语)。在ADSI的术语中,这些目录服务或者数据域可以称为名称空间。ADSI给我们提供了规范化的查询和修改这些名称空间数据的方法。

要理解ADSI,你可能得了解一些微软的Component Object Model(组件对象模型,COM),因为这是构造ADSI的基础。市面上有不少关于COM的书,不过我们可以把COM的精髓提炼为下面这些要点:

- 我们要使用COM来处理的所有东西都是对象。^[注19]
- 对象提供了接口(也就是一组方法,用来和对象交互)。从Perl中,我们可以使用从IDispatch接口提供的(或继承的)方法。好在大多数的ADSI接口(或其子接口)提供的方法都是从IDispatch继承而来,如IADsUser、IADsComputer、IADsPrintQueue等等。
- 对象封装的值,也就是那些可以通过方法查询或者修改的东西,被称为属性。我们在这一章会介绍两种属性:接口属性(也就是通过接口定义的属性)以及模式(schema)属性(也就是通过模式对象定义的属性,稍后会详细介绍)。大多数场合我们提到属性时指的都是接口属性,除非明确说明是“接口属性”。

以上都只是面向对象编程中常见的术语,不过等到ADSI/COM术语和其他面向对象术语混在一起时就会比较麻烦了。

比如在ADSI中,我们会讨论两种不同类型的对象:叶(leaf)和容器(container)。其中叶对象封装的是真正的数据,而容器对象只是用来容纳(或包含)其他对象。从LDAP角度来看,最容易理解的方法就是把这两个概念映射为“条目”和“分支点”。从一个角度来看,我们在讨论对象和相关属性;而从另一个角度来说,其实谈的就是条目和属性。那么如何统一这个分歧呢,因为其实我们说来说去不过是在谈同样的数据而已?

可以这样想:LDAP服务器才是条目和相关属性的真正存储者。不过在使用ADSI(而不是直接使用LDAP)来访问树中条目的时候,ADSI会帮你从LDAP服务器中查找数据,然后经过几层封装后以COM对象的方式展现给你。你使用必要方法来获得的值现在称为

注19: COM其实是一个用来和Object Linking and Embedding(对象链接和嵌入,OLE)这个大平台中的对象通信的协议。在这一节我会尽可能避免引入太多的微软术语,不过如果你有兴趣深入了解,可以参考<http://www.microsoft.com/com>。

“属性”。如果你修改了对象的属性，那么当你把对象归还给ADSI的时候，它会帮你完成相应LDAP数据的存储。

可能有人会问：“那么为什么不直接用LDAP和服务器交互呢？”答案如下：

- 一旦我们能够和一种支持ADSI的目录服务交互，那么我们也可以和其他支持ADSI的目录服务交互。
- ADSI的封装能让目录服务编程容易一些。
- 微软让我们使用ADSI。使用微软支持的编程接口几乎总是没错的。

要开始Perl的ADSI编程，我们必须引入*ADsPath*。*ADsPath*提供了唯一识别名称空间里任何对象的方式，它看上去如下：

```
<progID>:<path to object>
```

这里的<progID>是供应者在程序级的标识符，而<path to object>则是对供应者来说有效定位名称空间对象的方式。最常见的两个progID是LDAP和WinNT（WinNT也就是第3章中的SAM数据库）。

下面就是ADSI SDK文档中列出的*ADsPath*范例：

```
WinNT://MyDomain/MyServer/User
WinNT://MyDomain/JohnSmith,user
LDAP://ldapsvr/CN=TopHat,DC=DEV,DC=MSFT,DC=COM,0=Internet
LDAP://MyDomain.microsoft.com/CN=TopH,DC=DEV,DC=MSFT,DC=COM,0=Internet
```

这些*ADsPath*看上去很像URL，这并不是偶然的，因为URL和*ADsPath*都是用来完成同样的使命：以一种不带歧义的方式指明在某种数据服务中的数据。对于LDAP *ADsPath*来说，我们其实就是在使用RFC（编号2255）中规定的LDAP URL格式（参考附录C）。

警告：要注意<progID>部分是区分大小写的。把它错误地写成winnt、ldap或者WINNT都会导致错误，唯一正确的写法是WinNT和LDAP。另外要注意的是某些字符不能直接在*ADsPath*中使用，必须使用反斜线转义（或用16进制表示）^[注20]。目前来说，这些字符包括回车符、换行符、,、;、"、#、+、<、=、>以及\。

在讨论WinNT和LDAP两个名称空间的时候我们还会进一步深入介绍它们的*ADsPath*。现在我们先看看在Perl程序中如何使用ADSI。

注20：曾经有一个关于病人看医生的笑话，说的是病人对医生抱怨说“我的胳膊每次做这个动作的时候就会疼”，结果医生回答说“那么就别做这个动作了！”这里我也得作出同样的建议，不要使用这些特殊字符来构造*ADsPath*，那样做只是自找麻烦。

ADSI常用工具

任何运行Windows 2000或以上版本的机器上都已经有了内置于OS中的ADSI。我建议你在<http://www.microsoft.com/adsi>搜索并下载ADSI SDK，因为这里面有ADSI说明文档和一个好用的ADSI对象浏览器，名叫ADsVW。这个SDK中也带有各种语言的ADSI编程范例，包括Perl版本的。不过目前的ADSI发行包中的范例还是依赖于老掉牙的OLE.pm模块，所以虽然其中有些可以参考的地方，但还是有些局限性的。从这个链接出发你应该能找到很多重要的ADSI文档，包括adsi25.chm，这个压缩的HTML帮助文件里面包含了目前能找到的最佳ADSI文档。

在开始编程之前，最好能从<http://public.activestate.com/authors/tobyeverett/>下载Toby Everett用Perl写的ADSI对象浏览器。这个工具能带你浏览ADSI名称空间。请尽早访问这个站点，它能让你ADSI编程生涯更加顺利。虽然这个站点有些日子没有更新了，不过它仍然是使用Perl进行ADSI编程的最佳起点。

最后还有一个提示，最好能熟悉VBScript，以便读懂那些使用这个语言编写的脚本。虽然这个语言可能让你反胃，但你越深入ADSI，应该就会读到越多的VBScript代码。附录F以及本章末尾的参考资料部分列出的参考资料应该能加速你的学习过程。

从Perl使用ADSI

Jan Dubois维护的Win32::OLE系列模块给Perl和ADSI之间架设了桥梁，因为ADSI是基于OLE的COM实现的。在载入主要的模块之后，我们可以这样获得一个ADSI对象：

```
use Win32::OLE;

$adsobj = Win32::OLE->GetObject($AdsPath) or
    die "Unable to retrieve the object for $AdsPath\n";
```

注意：下面的两个提示能帮你避免不必要的惊讶。首先，如果你在Perl调试器中运行下面两行代码，然后查看返回的对象引用，可能会看到下面的信息：

```
DB<3> x $adsobj
0 Win32::OLE=HASH(0x10fe0d4)
    empty hash
```

别担心。Win32::OLE能很好地支持tie变量。这个对象看上去是空的，但只要正确使用，就能神奇地返回正确信息。

另外，如果GetObject调用返回类似下面的信息（尤其是在调试器中运行时）：

```
Win32::OLE(0.1403) error 0x8007202b:  
"A referral was returned from the server"
```

这往往意味着你提供的ADsPath对于LDAP供应者来说并不存在。这时候最好查看是否在ADsPath中有输入错误，比如你可能把LDAP://dc=example,dc=com输入成了LDAP://dc=example,dc=com。

Win32::OLE->GetObject()能返回OLE *moniker*（也就是对象的唯一标识符，这里是ADsPath）对应的ADSI对象。这个调用还能完成对象的绑定过程，这个过程在之前的LDAP相关章节已经介绍过。默认情况下我们会用执行脚本的用户的身份来完成绑定。

这里我们使用Perl的哈希引用语法来访问ADSI对象的接口属性值：

```
$value = $adsobj->{key}
```

比如对象有一个Name属性定义在接口中（这个属性总是有的），你就可以这样来获取它：

```
print $adsobj->{Name}."\n";
```

可以这样设置接口属性的值：

```
$adsobj->{FullName}= "Oog"; # 在缓存中设置属性
```

ADSI对象的属性存储在一块名为*property cache*的内存缓存里面。第一次访问对象的属性时这个缓存将被填充。之后再次访问同样的属性时，会直接从这个缓存里面取值，不必再次查询目录服务。如果你想手动填充这个缓存，可以调用对象的GetInfo()或者GetInfoEx()方法（GetInfo()的增强版），马上能看到范例。

因为第一次访问的时候会自动调用它们，所以GetInfo()和GetInfoEx()往往不为人知，不过有时候确实会需要显式调用它们（虽然我们并不会在本书中展示范例）。下面有两种可能的情况：

1. 有的对象属性必须通过显式调用GetInfoEx()才能获取。比如Microsoft Exchange 5.5的LDAP供应者的属性大多必须通过GetInfoEx()调用才能访问。可以参考<http://public.activestate.com/authors/tobyeverett/>了解更多特殊情况。
2. 如果你的目录是多人可写的，那么某个刚刚取得的对象可能已经被其他人修改过了。这时候对象的属性缓存就会含有过时的信息，可以通过GetInfo()和GetInfoEx()来刷新缓存。

要通过ADSI完成对后端目录服务和数据源的更新，在完成对象修改之后，必须调用特

殊的SetInfo()方法。SetInfo()能刷新属性缓存的修改，使得目录服务或数据源得到更新。

对某个ADSI对象实例进行方法调用很简单：

```
$adsobj->Method($arguments...)
```

所以，在修改了对象的属性之后，我们可以用这一行代码来刷新修改：

```
$adsobj->SetInfo();
```

这样就能刷新属性缓存的数据到底层目录服务或其他数据源中。

一个常用的Win32::OLE调用是Win32::OLE->LastError()。这个调用能返回最近的OLE操作产生的错误信息。Perl运行时的-w开关（比如perl -w script）也能使得最近一次OLE错误的详细信息被自动打印出来。这些错误信息往往是调试时的主要线索，所以请好好使用它们。

目前看到的ADSI代码还是非常标准的Perl程序，因为这只是一些泛泛的使用。下面我们会展示一些比较特殊的案例。

处理容器对象/集合对象

之前我们提到过有两种类型的ADSI对象：叶对象和容器对象。叶对象代表纯数据，而容器对象（也可以按照OLE/COM的术语称为“集合对象”）用来容纳其他对象。在ADSI中识别它们的另一个方法是：叶对象没有子对象，而容器对象有。

容器对象需要特殊处理，因为我们大多数时候只是对它们的子对象里的数据感兴趣。有两种方式从Perl来访问这些子对象。Win32::OLE提供了一个特殊的名叫in()的函数，不过这个函数在模块载入时并不会自动导入，所以需要显式导入：

```
use Win32::OLE qw(in);
```

in()能返回容器包含的子对象列表的引用。这让我们可以很容易写出下面这样的代码：

```
foreach $child (in $adsobj){  
    print $child->{Name}  
}
```

另外，我们还可以用Win32::OLE的子模块Win32::OLE::Enum。使用Win32::OLE::Enum->new()，我们就能从容器对象创建一个枚举对象：

```
use Win32::OLE::Enum;  
  
$enobj = Win32::OLE::Enum->new($adsobj);
```

然后我们可以对这个枚举对象调用一些方法来获得\$adsobj的子对象。\$enobj->Next()能够返回下一个子对象，而如果给这个方法一个可选参数就可以返回下N个子对象。\$enobj->All()则会返回子对象列表的引用。Win32::OLE::Enum还提供了其他方法（可以参考模块说明文档），不过这些方法是最常用的。

识别容器对象

无法先验地知道某个对象是否是容器对象。所以Perl也无法预先得知对象的“容器性”。最接近的识别方法就是尝试创建一个枚举对象，在失败的时候就能知道它并非容器对象。下面的代码就是相应的实现：

```
use Win32::OLE;
use Win32::OLE::Enum;

eval {$enobj = Win32::OLE::Enum->new($adsobj)};
print 'object is ' . ($@ ? 'not ' : '') . "a container\n";
```

另外你还可以查看对象在其他信息源中的描述。这正是我们的第三种特殊案例。

那么如何了解关于对象的其他信息？

直到目前为止我们还没有讨论一个重要问题。我们之前在两个名称空间中访问对象属性的时候，并没有告诉大家在哪里能找到这个对象的属性列表。那么这些信息是从哪里获得的呢？有没有办法程序化处理这些属性呢？

对于这个问题并没有标准答案，不过我们可以介绍几个常规思路。第一个可以查找的资料就是之前介绍的ADSI说明文档，尤其是刚才在补充内容“ADSI常用工具”中介绍的帮助文件。这个文件里面有丰富的帮助信息，对于这个问题来说，应该从Active Directory Service Interfaces 2.5→ADSI Reference→ADSI System Providers这个位置来开始查看。

在有些情况下，这个文档是唯一的方法信息来源，不过这里我们会尝试另一个更有趣的方式：可以通过ADSI元数据来获取属性名。这就是我们一直提到的模式（schema）属性的实际应用了，如果你不记得在哪里提到过它，请参考“ADSI基础”一节的开头部分。

每个ADSI对象都有一个叫做Schema的属性，它指明了通往对象的模式对象的ADsPath。比如下面的代码：

```
use Win32::OLE;

$ADsPath = 'WinNT://BEESKNEES,computer';
$adsobj = Win32::OLE->GetObject($ADsPath) or
    die "Unable to retrieve the object for $ADsPath\n";
print 'This is a ' . $adsobj->{Class} . "object, schema is at:\n".
    $adsobj->{Schema} . "\n";
```

就能打印：

```
This is a Computer object, schema is at:  
WinNT://DomainName/Schema/Computer
```

`$adsobj->{ Schema }`的值是描述Computer类的模式对象的ADsPath。这里我们提到的“模式”也就是LDAP的模式。在LDAP中，模式定义了那些对于对象类来说是必需的属性。在ADSI中，模式对象也存储了同样的关于对象类及其模式属性的信息。

如果你感兴趣的是某个对象的可用属性名，那么你可以查看这个模式对象的两个属性：`MandatoryProperties`和`OptionalProperties`。让我们把刚才例子中的`print`语句改成如下语句：

```
$schmobj = Win32::OLE->GetObject($adsobj->{Schema}) or  
die "Unable to retrieve the object for $ADsPath\n";  
print join("\n",@{$schmobj->{MandatoryProperties}},  
           @{$schmobj->{OptionalProperties}}), "\n";
```

这次打印的信息是：

```
Owner  
Division  
OperatingSystem  
OperatingSystemVersion  
Processor  
ProcessorCount
```

现在我们知道了在WinNT名称空间中的Computer对象的可用模式接口属性名。看来很有用，不是吗？

模式属性的获取和设置方法与接口属性稍有不同。之前提到，接口属性可以这样获取和设置：

```
# 获取并设置 INTERFACE 属性  
$value = $obj->{property};  
$obj->{property} = $value;
```

模式属性是使用特殊的方法获取和设置的：

```
# 获取并设置 SCHEMA 属性  
$value = $obj->Get('property');  
$obj->Put('property', 'value');
```

之前介绍的那些获取和设置接口属性的机制对模式属性也同样适用（即属性缓存、`SetInfo()`等等）。除了获取和设置使用的方法以外，还有一个地方可以区分模式属性和接口属性，那就是它们的名字。有时候某个对象的同一个属性可以有两个不同的名字：一个是接口属性的名字，另一个则是模式属性的名字。例如下面两行代码都能取得同一个用户属性：

```
$len = $userobj->{PasswordMinimumLength}; # 按接口属性
$len = $userobj->Get('MinPasswordLength'); # 按模式属性
```

两种属性并存的现状是因为接口属性来自于底层的COM模型。当开发人员定义接口的时候，同时也定义了接口属性。如果他们后来需要扩展属性，那么必须同时修改COM接口以及使用接口的代码。作为对比，模式属性的扩展则不需要修改COM接口。作为程序员，应该能自如处理两种属性，因为有时候某些对象数据只能通过其中某种属性获得。

从实际操作角度来考虑，如果你希望了解某个对象类的接口属性或模式属性，而又懒得写程序来获取。那么我推荐你使用Toby Everett的ADSI浏览器，这在之前提到过。图9-2所示是这个浏览器使用时的截图。

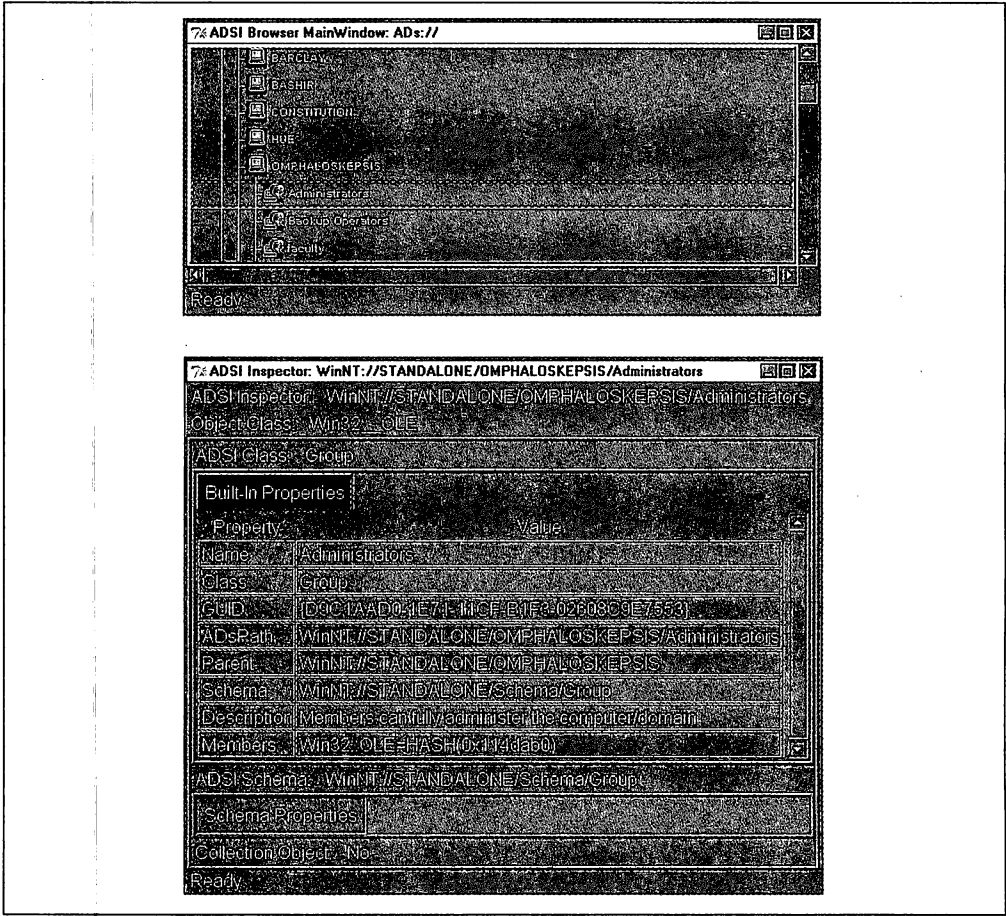


图9-2: Everett的ADSI浏览器显示了一个管理员组对象

另外，SDK示例的General子目录里面有一个名叫ADSIDump的程序，它能打印ADSI树的所有内容。

搜索

最后一个特殊案例是关于搜索。之前在“LDAP：一种复杂的目录服务”一节中，我们花了不少篇幅介绍LDAP，不过到了ADSI，则没有讨论这个话题。这是因为从Perl来进行ADSI搜索是非常痛苦的（其实所有使用OLE自动化接口的语言都一样），尤其是子树搜索或者复杂的过滤器搜索。复杂搜索的问题在于它们必须通过ADSI框架以外的机制来完成（不想提那些奇怪的微软术语）。

不过系统管理员都是一些以苦为乐的人，所以我们还是得介绍。我们可以从简单搜索开始介绍，这种查找某个对象（使用base范围）或查找某个对象的子对象（使用one范围）的搜索可以用Perl来实现：

- 对于单个对象的搜索，可以直接取得感兴趣的属性并且使用Perl的常规比较操作符来判断是否匹配：

```
if ($adsobj->{cn} eq 'Mark Sausville' and $adsobj->{State} eq 'CA'){...}
```

- 对于某个对象的子对象的搜索，可以使用刚才介绍过的容器对象的访问机制来获取子对象并且逐个判断是否匹配。我们稍后会展示这类搜索。

如果你打算进行更复杂的搜索，比如搜索整个目录树（或子树），那么必须使用某种叫做Active X数据对象（ActiveX Data Objects, ADO）的“中间件”技术。ADO为脚本编程语言访问微软的OLE DB打通了渠道。OLE DB给关系数据库和目录服务这些数据源提供了通用面向数据库的接口。对我们来说，需要使用ADO来与ADSI交互，而ADSI会与真正的目录服务交互。因为ADO是一种面向数据库的方法论，所以下面的代码会与之前在第7章讨论的ODBC连接方法有关。

注意：ADO只能用来访问LDAP ADSI供应者，它不能用来查询WinNT名称空间。

ADO是一个独立的主题，只是与目录服务有些许关系。所以我们只会介绍ADO的范例代码，只有与ADSI相关的部分才给予解释。要想了解更多关于ADO的信息，可以在微软的网站上搜索“ADO”，也可以查看这个维基页面：http://en.wikipedia.org/wiki/ActiveX_Data_Objects。

下面的代码能列出指定域内所有组的名称：

```
use Win32::OLE qw(in);

# 取得 ADO 对象，设置提供者，打开连接
$c = Win32::OLE->new('ADODB.Connection');
$c->{Provider} = 'AdDSOObject';
```



```

$c->Open('ADSI Provider');
die Win32::OLE->LastError() if Win32::OLE->LastError();

# 准备并执行查询
$ADsPath = 'LDAP://ldapserver/dc=example,dc=com';
$rs = $c->Execute("<$ADsPath>;(objectClass=Group);Name;SubTree");
die Win32::OLE->LastError() if Win32::OLE->LastError();

until ($rs->EOF){
    print $rs->Fields(0)->{Value}, "\n";
    $rs->MoveNext;
}

$rs->Close;
$c->Close;

```

在载入模块之后，这段代码获得了一个ADO Connection对象实例，然后设置对象实例的供应者，最后用此供应者打开连接。这个连接是用执行脚本的用户的身份打开的，不过我们也可以通过对象属性来设置，从而以其他用户身份打开。

然后我们使用Execute()来执行搜索。这个搜索可以用SQL或者ADSI“方言”（dialect）指定^[注21]。ADSI方言使用以分号分隔的一个由4个参数组成的命令字符串。

警告：请仔细对待ADSI ADO供应者的一个怪癖，在查询时分号的周围不能有空格，否则会失败。

参数包括：

- 一个设置服务器和搜索基DN的ADsPath（在尖括号中）
- 一个搜索过滤器（这个语法和之前介绍过的LDAP过滤器一致）
- 需要返回的属性名（也可以是属性名列表，用逗号分隔）
- 搜索的范围：Base、OneLevel或者SubTree（根据LDAP标准）

Execute()返回一个对查询返回的第一个ADO RecordSet对象的引用。于是我们从Fields()方法来获得属性列表，然后用Value属性返回具体值。这里我们的查询只会返回一个值，就是搜索的Group对象名。下面是在Windows Server 2003上的输出节选：

```

Domain Computers
Domain Users
RAS and IAS Servers
Users

```

注21：一旦提到SQL就让人联想到一个有趣的话题：可以配置Microsoft SQL Server来访问ADSI供应者，而不一定要访问常规数据库。这意味着你可以使用标准SQL Server的SQL查询来查找ADSI的ActiveDirectory对象，而不是访问数据库。这真的很酷。

```
Domain Guests
Group Policy Creator Owners
Enterprise Admins
Server Operators
Account Operators
Print Operators
Replicator
Domain Controllers
Schema Admins
Remote Desktop Users
Network Configuration Operators
Incoming Forest Trust Builders
Performance Monitor Users
Terminal Server License Servers
Pre-Windows 2000 Compatible Access
Performance Log Users
Windows Authorization Access Group
Backup Operators
Domain Admins
Administrators
Cert Publishers
Guests
DnsAdmins
DnsUpdateProxy
Debugger Users
```

使用WinNT和LDAP名称空间执行常规管理任务

现在我们已经介绍了所有的特殊案例，下面可以使用ADSI进行一些日常管理了。这里的重点是要让你体会到ADSI信息可以支持的任务，以便你将来可以在此基础上进一步开发其他脚本。

我们逐个介绍两个名称空间。首先是WinNT名称空间，使用它可以获得对Windows本地的SAM数据库的访问，其中含有本地用户、组、打印机和服务之类的信息。

第二个是老朋友LDAP名称空间。在Windows 2000之后LDAP成为主要的信息来源。大多数WinNT名称空间的对象也可以通过LDAP来访问。不过哪怕到了Windows Server 2003，还是有些任务必须通过WinNT名称空间来完成，比如创建本地主机账户。

处理两个名称空间的代码看上去很相似（毕竟这是我们使用ADSI的主要动力），不过你还是应该注意两个重要的区别。首先，ADsPath的格式有些不同。WinNT ADsPath的格式在ADSI SDK中规定了，必须是下面的格式之一：

```
WinNT://DomainName[/ComputerName[/ObjectName[,className]]]
WinNT://DomainName[/ObjectName[,className]]
WinNT://ComputerName,computer]
WinNT:
```

而LDAP ADsPath格式是：

```
LDAP://HostName[:PortNumber][/]DistinguishedName]
```

请注意LDAP和WinNT名称空间的对象属性大体是一致的，但并不完全相同。比如，我们可以从两个名称空间访问同一个用户对象，但是只有在LDAP名称空间才能访问与活动目录相关的用户属性。

要特别注意这两个名称空间的模式（schema）上的差异。比如WinNT的User类没有必备属性，而LDAP User类有必备属性。使用LDAP名称空间，你至少需要填写cn和samAccountName属性才能成功创建一个User对象。

搞清楚了这些差异之后，让我们来看看实际代码。为节省篇幅，我们会忽略大多数的错误检查，不过你可以用-w开关来运行脚本，也可以在代码中根据需要插入下面这行代码：

```
die 'OLE error :'.Win32::OLE->LastError() if Win32::OLE->LastError();
```

注意：在下面的范例中，你会发现我在WinNT和LDAP名称空间里辗转反复，这是为了让你熟悉这两者。具体任务中选择使用哪个？这主要是看任务的类型和活动目录的大小。

有的时候不需要选择。比如在处理本地用户、服务和打印队列的时候必须选择WinNT。相应地，那些只有在AD中才有的用户属性、AD控制对象等等必须使用LDAP。

对于其他任务来说，则可以自由选择。大多数时候建议你选择LDAP，因为这会更加有效，当然也会更加复杂一点。在使用LDAP名称空间的时候，可以直接对AD树中深层次的对象直接操作，不必像在WinNT中那样枚举许多的父对象。如果你的AD树相对简单，那么这个优势就没那么明显，可能使用WinNT更加方便。这就是主要看你自己决定的场合。

通过ADSI访问用户信息

要通过WinNT名称空间来列出用户：

```
use Win32::OLE qw(in);

# 'WinNT://CurrentComputername,computer' ——当前计算机上的本地用户
# 'WinNT://DCname, computer' ——当前域下客户机上的用户
# 'WinNT://DomainName/DCName,computer' ——指定某个域

my $ADsPath= 'WinNT://DomainName/DCName,computer';
my $c = Win32::OLE->GetObject($ADsPath) or die "Unable to get $ADsPath\n";
foreach my $adsobj (in $c){
    print $adsobj->{Name},"\n" if ($adsobj->{Class} eq 'User');
}
```

如果你想通过LDAP名称空间来完成对整个目录树中所有用户的搜索，那么可以使用前一节介绍的基于ADO的搜索机制。

要创建一个本地用户并设置用户的全名：

```
use Win32::OLE;

my $ADsPath='WinNT://LocalMachineName,computer';
my $c = Win32::OLE->GetObject($ADsPath) or die "Unable to get $ADsPath\n";

# 创建并返回一个 User 对象
my $u = $c->Create('user',$username);
$u->SetInfo(); # 修改之前必须先创建该用户对象

# WinNT 名称空间中不允许出现空格，所以“Full”和“Name”是连起来写的
$u->{FullName} = $fullname;
$u->SetInfo();
```

通过 LDAP 名称空间创建（活动目录级别的）全局用户的代码如下：

```
use Win32::OLE;

# 这会创建位于目录树的 cn=Users 分支下的用户。
# 如果要修改为其他分支下的用户，仅需修改以下这行
my $ADsPath= 'LDAP://ldapserver,CN=Users,dc=example,dc=com';

my $c = Win32::OLE->GetObject($ADsPath) or die "Unable to get $ADsPath\n";

# 创建并返回一个 User 对象
my $u=$c->Create('user','cn='.$commonname);
$u->{samAccountName} = $username;
# 重要：修改之前必须先创建该用户对象
$u->SetInfo();

# LDAP 名称空间中需要使用空格，所以“Full”和“Name”中间有空格（哎）
$u->{'Full Name'} = $fullname;
$u->SetInfo();
```

删除本地用户只需要小小的代码改动而已：

```
use Win32::OLE;

my $ADsPath= 'WinNT://DomainName/ComputerName,computer';
my $c = Win32::OLE->GetObject($ADsPath) or die "Unable to get $ADsPath\n";

# 删除 User 对象；注意，最后还是要更新容器对象
$c->Delete('user',$username);
$c->SetInfo();
```

修改用户的密码只是调用一个方法而已：

```

use Win32::OLE;

# 或者 'LDAP://cn=$username,ou=staff,ou=users,dc=example,dc=com' (举例)
my $ADsPath= 'WinNT://DomainName/ComputerName/.$username;
my $u = Win32::OLE->GetObject($ADsPath) or die "Unable to get $ADsPath\n";

$u->ChangePasssword($oldpassword,$newpassword);
$u->SetInfo();

```

通过ADSI访问组

你可以简单修改一下用户枚举代码，从而实现通过WinNT名称空间来枚举可用用户组。需要改动的代码只有一行：

```

print $adsobj->{Name}, "\n" if ($adsobj->{Class} eq 'Group');

```

如果你希望通过LDAP名称空间来枚举用户组，最好使用ADO（参考“搜索”一节）。

创建和删除用户组的方法和刚才用户的创建、删除方法一致，区别只是第一个参数不同（必须修改成'group'）。比如：

```

my $g = $c->Create('group',$groupname);

```

在创建组之后如何把用户加入这个组（组名为GroupName）呢？

```

use Win32::OLE;

my $ADsPath= 'WinNT://DomainName/GroupName,group';

my $g = Win32::OLE->GetObject($ADsPath) or die "Unable to get $ADsPath\n";

# 使用 ADsPath 指定要操作的用户对象
$g->Add($userADsPath);

```

对于WinNT名称空间来说，之前提到的处理本地用户和全局用户的区别还是存在的。如果我们要把域用户加入组里，我们的\$userADsPath必须是域用户的DC。如果我们希望使用LDAP名称空间来完成此任务，我们的组必须以目录树的形式提供：

```

my $ADsPath= 'LDAP://cn=GroupName,ou=Groups,dc=example,dc=com';

```

从某个组中删除用户：

```

$c->Remove($userADsPath);

```

通过ADSI处理文件共享

现在我们开始完成一些更加有趣的ADSI任务。可以通过ADSI来指定某台机器共享它的一部分本地存储给其他机器：

```
use Win32::OLE;

my $ADsPath= 'WinNT://ComputerName/lanmanserver';

my $c = Win32::OLE->GetObject($ADsPath) or die "Unable to get $ADsPath\n";

my $s = $c->Create('fileshare',$sharename);
$s->{path} = 'C:\directory';
$s->{description} = 'This is a Perl created share';
$s->SetInfo();
```

通过Delete()方法可以删除文件共享。

注意：在我们转移到其他任务之前，先容我再次向你推销SDK说明文档，因为多读说明文档你常常能发现惊喜。在ADSI 2.5的帮助文件中，跟随Active Directory Service Interfaces 2.5→ADSI Reference→ADSI Interfaces→Persistent Object Interfaces→IADsFileShare这条路线，你就能看到fileshare对象有一个CurrentUserCount属性，其中包含了目前连接到此文件共享的用户数。这真是一个非常有用的属性。

通过ADSI处理打印队列和打印任务

下面展示了如何确定某台机器上的打印队列的名字以及服务于队列的打印机模型：

```
use Win32::OLE qw(in);

my $ADsPath='WinNT://DomainName/PrintServerName,computer';

my $c = Win32::OLE->GetObject($ADsPath) or die "Unable to get $ADsPath\n";

foreach my $adsobj (in $c){
    print $adsobj->{Name}.' : '. $adsobj->{Model}."\n"
        if ($adsobj->{Class} eq 'PrintQueue');
}
```

一旦有了打印队列的名字，就可以直接连接这个队列以查询并且控制它：

```
use Win32::OLE qw(in);

# 该表取自 ADSI 2.5 SDK中的以下部分：
# 'Active Directory Service Interfaces 2.5->ADSI Reference->
# ADSI Interfaces->Dynamic Object Interfaces->IADsPrintQueueOperations->
# IADsPrintQueueOperations Property Methods' (啊)
```

```

my %status =
  (0x00000001 => 'PAUSED',           0x00000002 => 'PENDING_DELETION',
   0x00000003 => 'ERROR',             0x00000004 => 'PAPER_JAM',
   0x00000005 => 'PAPER_OUT',         0x00000006 => 'MANUAL_FEED',
   0x00000007 => 'PAPER_PROBLEM',     0x00000008 => 'OFFLINE',
   0x00000100 => 'IO_ACTIVE',          0x00000200 => 'BUSY',
   0x00000400 => 'PRINTING',           0x00000800 => 'OUTPUT_BIN_FULL',
   0x00001000 => 'NOT_AVAILABLE',      0x00002000 => 'WAITING',
   0x00004000 => 'PROCESSING',         0x00008000 => 'INITIALIZING',
   0x00010000 => 'WARMING_UP',         0x00020000 => 'TONER_LOW',
   0x00040000 => 'NO_TONER',           0x00080000 => 'PAGE_PUNT',
   0x00100000 => 'USER_INTERVENTION',  0x00200000 => 'OUT_OF_MEMORY',
   0x00400000 => 'DOOR_OPEN',          0x00800000 => 'SERVER_UNKNOWN',
   0x01000000 => 'POWER_SAVE');

my $ADsPath = 'WinNT://PrintServerName/PrintQueueName';

my $p = Win32::OLE->GetObject($ADsPath) or die "Unable to get $ADsPath\n";

print 'The printer status for ' . $c->{Name} . ' is ' .
  ((exists $p->{status}) ? $status{$c->{status}} : 'NOT ACTIVE') . "\n";

```

PrintQueue对象提供了一组你所需要的打印队列控制方法：Pause()、Resume()和Purge()，这使得我们能够轻松处理队列。不过如果我们要处理的是队列中某个具体的打印任务呢？

要获得某个具体任务，你可以通过调用PrintQueue对象的PrintJobs()方法。这个方法会返回一组PrintJob对象，而这些对象有自己的属性和方法。比如下面的代码就能显示特定队列中的打印任务：

```

use Win32::OLE qw(in);

# 该表取自 ADSI 2.5 SDK中的以下部分：
# 'Active Directory Service Interfaces 2.5->ADSI Reference->
# ADSI Interfaces->Dynamic Object Interfaces->IADsPrintJobOperations->
# IADsPrintJobOperations Property Methods' (嗨，嗨)

my %status = (0x00000001 => 'PAUSED', 0x00000002 => 'ERROR',
              0x00000004 => 'DELETING', 0x00000010 => 'PRINTING',
              0x00000020 => 'OFFLINE', 0x00000040 => 'PAPEROUT',
              0x00000080 => 'PRINTED', 0x00000100 => 'DELETED');

my $ADsPath = 'WinNT://PrintServerName/PrintQueueName';

my $p = Win32::OLE->GetObject($ADsPath) or die "Unable to get $ADsPath\n";

$jobs = $p->PrintJobs();
foreach my $job (in $jobs){
  print $job->{User} . "\t" . $job->{Description} . "\t" .
    $status{$job->{status}} . "\n";
}

```

这里的每一个任务也都可以接受Pause()和Resume()命令。

通过ADSI来处理基于Windows的操作系统服务

最后一组例子是关于如何定位Windows机器上的服务，以及如何启动、停止它们。如同这一章其他代码，下面的代码都必须以高权限账户才能运行。

要列出机器上的服务及其状态，我们可以使用下面的代码：

```
use Win32::OLE qw(in);

# 该表取自 ADSI 2.5 SDK中的以下部分：
# 'Active Directory Service Interfaces 2.5->ADSI Reference->
# ADSI Interfaces->Dynamic Object Interfaces->IADsServiceOperations->
# IADsServiceOperations Property Methods'

my %status =
  (0x00000001 => 'STOPPED',          0x00000002 => 'START_PENDING',
   0x00000003 => 'STOP_PENDING',      0x00000004 => 'RUNNING',
   0x00000005 => 'CONTINUE_PENDING', 0x00000006 => 'PAUSE_PENDING',
   0x00000007 => 'PAUSED',            0x00000008 => 'ERROR');

my $ADsPath = 'WinNT://DomainName/ComputerName,computer';

my $c = Win32::OLE->GetObject($ADsPath) or die "Unable to get $ADsPath\n";

foreach my $adsobj (in $c){
  print $adsobj->{DisplayName} . ':' . $status{$adsobj->{status}} . "\n"
    if ($adsobj->{Class} eq 'Service');
}
```

要启动、停止、暂停或者继续某个服务，我们可以调用相应的方法（Start()、Stop()等等）。下面的代码能启动Windows机器上的Network Time服务（如果它是停止的）：

```
use Win32::OLE;

my $ADsPath = 'WinNT://DomainName/ComputerName/W32Time,service';

my $s = Win32::OLE->GetObject($ADsPath) or die "Unable to get $ADsPath\n";

$s->Start();
# 最好在此处检查实际状态以作确认，如果没有启动就再次循环发出启动命令直到成功
```

为避免用户名和机器名的冲突，以上代码也可以写成这样：

```
use Win32::OLE;

my $d = Win32::OLE->GetObject('WinNT://Domain');
my $c = $d->GetObject('Computer', $computername);
my $s = $c->GetObject('Service', 'W32Time');

$s->Start();
```

停止这个服务只需要修改最后一行：


```
$s->Stop();  
# 最好在此处检查实际状态以作确认，停顿一两秒后如果还没停止就重新发送停止命令直到成功
```

这些例子应该能开阔你的思路，让你明白通过Perl使用ADSI能完成多少系统管理任务。目录服务和其接口可以成为你的计算基础设施非常强大的部分。

本章所用模块

模块名	CPAN ID	版本
Net::Telnet	JROGERS	3.03
Net::Finger	FIMM	1.06
Net::Whois::Raw	DESPAIR	0.34
Net::LDAP	GBARR	0.32
Sys::Hostname (随 Perl 发布)		1.11
Win32::OLE (随 ActiveState Perl 发布)	JDB	0.17

更多参考资料

如果想了解本章中所讨论主题的深入信息，可以参考以下列出的资源。

RFC 1288: The Finger User Information Protocol，由D. Zimmerman所著（1991年），它定义了Finger。

<ftp://sipb.mit.edu/pub/whois/whois-servers.list>是大部分主要WHOIS服务器的列表。

RFC 954: NICNAME/WHOIS，由K. Harrenstien、M. Stahl和E. Feinler所著（1985年）定义了WHOIS。

LDAP

<http://ldap.perl.org>是Net::LDAP模块的主页。

<http://www.openldap.org>是OpenLDAP的主页，它是一个开发比较活跃的免费LDAP服务器。

JXplorer (<http://www.jxplorer.org>) 和Apache Directory Studio (<http://directory.apache.org/studio/>) 是两个不错的免费GUI LDAP浏览器，它们都能和我所使用过的所有的LDAP服务器一起正常工作。

led (<http://sourceforge.net/projects/led/>) 和ldapdiff (<https://launchpad.net/ldapdiff/>) 是两个很方便的用来编辑LDAP条目/树的命令行工具。前者会调用你所选择的编辑器来编

辑LDIF格式的条目，后者能帮助你显示实际LDAP树和LDIF文件之间的差异（如果你愿意，还能帮你生成补丁）。

你也许还会想要参考以下这些LDAP的相关资料：

- 《Implementing LDAP》，由Mark Wilcox所著（Wrox Press出版）
- 《LDAP-HOWTO》，由Mark Greennan所著（1999年），可以在<http://www.grennan.com/ldap-HOWTO.html>找到
- 《Understanding and Deploying LDAP Directory Services》（Second Edition），由Tim Howes等人所著（Addison-Wesley出版）
- *RFC 1823: The LDAP Application Program Interface*，由T. Howes和M. Smith所著（1995年）
- *RFC 2222: Simple Authentication and Security Layer (SASL)*，由J. Myers所著（1997年）
- *RFC 2251: Lightweight Directory Access Protocol (v3)*，由M. Wahl、T. Howes和S. Kille所著（1997年）
- *RFC 2252: Lightweight Directory Access Protocol (v3): Attribute Syntax Definitions*，由M. Wahl等人所著（1997年）
- *RFC 2254: The String Representation of LDAP Search Filters*，由T. Howes所著（1997年）
- *RFC 2255: The LDAP URL Format*，由T. Howes和M. Smith所著（1997年）
- *RFC 2256: A Summary of the X.500(96) User Schema for Use with LDAPv3*，由M. Wahl 所著（1997年）
- *RFC 2849: The LDAP Data Interchange Format (LDIF)—Technical Specification*，由Gordon Good所著（2000年）
- 《Understanding LDAP》，由Heinz Jonner等人所著（1998年），可以在<http://www.redbooks.ibm.com/abstracts/sg244986.html>找到（这是本很棒的介绍LDAP的“红宝书”）
- 《LDAP System Administration》，由Gerald Carter所著（O'Reilly出版）
- 《LDAP Programming, Management, and Integration》，由Clayton Donley所著（Manning出版）

ADSI

<http://cwwashington.netreach.net>是个很棒的（Perl无关的）关于ADSI脚本编程和其他一些微软技术的站点。

<http://msdn.microsoft.com/en-us/library/aa772170.aspx>是关于ADSI信息的权威资源。

<http://public.activestate.com/authors/tobyeverett/>包含Toby Everett收集的关于通过Perl使用ADSI的文档。

<http://www.15seconds.com>是另外一个很棒的（Perl无关的）关于ADSI脚本编程和其他微软技术的站点。

<http://isg.ee.ethz.ch/tools/realmen/>是一个几乎完全用Perl编写的Windows下完整的系统管理基础设施。

Robbie Allen是许多很棒的Windows和AD（Active Directory）书籍的作者/联合作者，他拥有一个网站<http://techtasks.com>，在这里你可以找到他书中所涉及的所有代码示例。可以毫不夸张地说，他的这些示例就是宝藏——这也是你能找到的关于ADSI编程的最有帮助的站点之一。关于Allen所做的更多贡献，请看第3章末尾的参考资料。

你也许还想查看以下资料：

- 《Active Directory》（Second Edition），由Alistair G. Lowe-Norris所著（O'Reilly出版）
- 《Managing Enterprise Active Directory Services》，由Robbie Allen和Richard Puckett所著（Addison-Wesley出版）
- 《Microsoft Windows 2000 Scripting Guide: Automating System Administration》（Microsoft Press出版）

日志文件

如果这不是本关于系统管理的书，那么用一整章来介绍日志文件是很罕见的。然而系统管理员和日志文件之间有着特殊关系。系统管理员就像可以和动物说话的Doolittle医生：能与一大堆软件和硬件交流。很多这种交流是通过日志文件来完成的，所以我们的系统管理员是日志文件专家。Perl在这方面可以帮上很大的忙。

我们不可能仅在一章里面涉及所有不同的针对日志的处理和分析方面的内容。关于统计和分析此类数据有其他整本书籍专门介绍，还有公司专门卖相关的帮助分析日志的产品。尽管如此，本章还是会展示一些通用方法和相关的Perl工具来激发和引导你在这方面的兴趣。

读取文本日志

日志有很多不同的种类，所以我们需要用不同的方式来处理他们。最常见类型的日志文件是完全由文本行组成的：流行的服务器包如Apache（Web服务器）、BIND（DNS服务器）及*sendmail*（E-mail服务器）都会产生大量的文本日志（尤其是在调试模式中）。Unix机器上绝大多数的日志看起来都很相似，因为它们都是由集中的日志工具*syslog*产生的。就我们而言，可以把*syslog*创建的文件当成和其他文本文件相同的文件。

下面是一个简单的Perl程序，用以扫描基于文本的日志文件中单词“error”出现的地方：

```
open my $LOG, '<', "$logfile" or die "Unable to open $logfile:$!\n";
while(my $line = <$LOG>){
    print if $line =~ /\berror\b/i;
}
close $LOG;
```

熟悉Perl的读者可能迫不及待想将上面的例子变成单行代码了，如下所示：

```
perl -ne 'print if /\berror\b/i' logfile
```

读取二进制日志文件

有时候写程序来处理日志文件也不是很容易的事。相对于易于解析的文本行，有些日志的记录方式产生的是比较晦涩的拥有专门格式的二进制文件，要想用单行Perl代码来解析它们基本上是不可能的。幸运的是，Perl不怕这些看起来比较诡异的文件。让我们看看几种可以用来处理这些文件的方式。我们将要针对的是两种不同的二进制文件范例：Unix的*wtmp*文件和基于Windows的操作系统事件日志。

在之前的第4章中，我们简要介绍了Unix主机的登录和注销的概念。在大多数Unix变种中，登录和注销这两个行为会被跟踪记录到在名为*wtmpx*（或者是*wtmp*）的文件中。通常，如果对某个用户的登录行为产生怀疑（比如说某个用户经常从哪台主机登录，但某次他从其他地方登录），我们一般会检查这个文件。在不同的操作系统上，这个文件所处的位置也可能不同（比如说，在Solaris上它在*/var/adm*中，在Linux上它在*/var/log*中^[注1]）。

在Windows上，事件日志所扮演的角色会更宽泛一些。它被用来集中记录所有实际上发生在这些机器上的活动，包括登录和注销、操作系统消息、安全事件等等。它的这个角色就类似于之前我们提到过的Unix上的*syslog*服务。

使用unpack()

Perl有个函数叫*unpack()*，它是被特别设计用来解析二进制和结构化数据的。让我们看看如何使用它来处理*wtmpx*文件。*wtmp*和*wtmpx*文件的格式在不同的Unix变种之间会有所不同。就这点，我们将介绍Solaris 10上的*wtmpx*文件和Linux 2.6上的*wtmp*文件。下面是Solaris 10上*wtmpx*文件的头两个记录的纯文本化的样子：

```
0000000 d n b \0 \0 \0 \0 \0 \0 \0 \0 \0 \0 \0 \0 \0 \0
0000020 \0 \0 \0 \0 \0 \0 \0 \0 \0 \0 \0 \0 \0 \0 \0
0000040 t s / 1 p t s / 1 \0 \0 \0 \0 \0 \0 \0 \0
0000060 \0 \0 \0 \0 \0 \0 \0 \0 \0 \0 \0 \0 \0 \0 \0
0000100 \0 \0 \0 \0 \0 \0 # 346 \0 007 \0 \0 \0 \0 \0 \0
0000120 D 9 . 253 \0 \t 313 234 \0 \0 \0 \0 \0 \0 \0 \0
0000140 \0 \0 \0 \0 \0 \0 \0 \0 \0 \0 \0 \0 \0 \0 \0
0000160 \0 ' p o o l - 1 4 1 - 1 5 4 - 1
0000200 2 1 - 5 . b o s . e a s t . v e
0000220 r i z o n . n e t \0 \0 \0 \0 \0 \0 \0 \0
0000240 \0 \0 \0 \0 \0 \0 \0 \0 \0 \0 \0 \0 \0 \0 \0
*
0000560 \0 \0 \0 T d n b \0 \0 \0 \0 \0 \0 \0 \0 \0 \0
0000600 \0 \0 \0 \0 \0 \0 \0 \0 \0 \0 \0 \0 \0 \0 \0
```

注1： 不过，Mac OS X拥有它自己的叫做Apple系统日志工具的日志框架，但它还是保留并同步更新*/var/run/utmpx*文件。

```

0000620 \0 \0 \0 \0 t s / 2 p t s / 2 \0 \0 \0
0000640 \0 \0 \0 \0 \0 \0 \0 \0 \0 \0 \0 \0 \0 \0
0000660 \0 \0 \0 \0 \0 \0 \0 \0 \0 \0 $ R \0 007 \0 \0
0000700 \0 \0 \0 \0 D 9 / 212 \0 016 L 315 \0 \0 \0 \0
0000720 \0 \0 \0 \0 \0 \0 \0 \0 \0 \0 \0 \0 \0 \0
0000740 \0 \0 \0 \0 \0 ' p o o l - 1 4 1 - 1
0000760 5 4 - 1 2 1 - 5 . b o s . e a s
0001000 t . v e r i z o n . n e t \0 \0 \0
0001020 \0 \0 \0 \0 \0 \0 \0 \0 \0 \0 \0 \0 \0 \0
*
0001340 \0 \0 \0 \0 \0 \0 \0 T

```

除非你已经熟悉了这种文件的结构，否则这种被称为“ASCII dump”的数据对你而言和乱码没什么区别。那么，我们该怎么去认识这种文件结构呢？

要了解这种文件格式最简单的方式就是查看读写该文件的程序的源代码。如果你不熟悉C语言，这个任务可能会让你感到气馁。幸运的是，实际上我们并不需要了解，甚至也不需要去查看大部分源代码，我们可以只查看定义了该文件格式的那部分内容就够了。

大部分读写*wtmp*文件的操作系统程序都会从一个较短的C包含文件中获取文件定义，这个文件一般是*/usr/include/utmp.h*或者*utmpx.h*。我们仅需要阅读拥有相关文件格式信息的C数据结构定义。如果你搜索`struct utmp {`，就能找到我们需要了解的部分。`struct utmp {`的下面几行定义了该结构中的各个字段。这些行应该各自都有符合C注释约定`/*text*/`的注释行来加以说明。

为了让你了解两个不同版本*wtmpx*之间的差异，我们来比较一下这两种不同操作系统上相关的代码内容片段。

下面是Solaris 10上*utmpx.h*的相关片段：

```

/*
 * This data structure describes the utmp *file* contents using
 * fixed-width data types. It should only be used by the implementation.
 *
 * Applications should use the getutxent(3c) family of routines to interact
 * with this database.
 */

struct futmpx {
    char    ut_user[32];           /* user login name */
    char    ut_id[4];              /* inittab id */
    char    ut_line[32];           /* device name (console, lnxx) */
    pid32_t ut_pid;                /* process id */
    int16_t ut_type;               /* type of entry */
    struct {
        int16_t e_termination;    /* process termination status */
        int16_t e_exit;           /* process exit status */
    } ut_exit;                     /* exit status of a process */
    struct timeval32 ut_tv;         /* time entry was made */

```

```

    int32_t ut_session;          /* session ID, user for windowing */
    int32_t pad[5];             /* reserved for future use */
    int16_t ut_syslen;          /* significant length of ut_host */
    char    ut_host[257];       /* remote host name */
};

```

下面这个片段来自Linux 2.6的*bits/utmp.h*文件：

```

struct utmp
{
    short int ut_type;          /* Type of login. */
    pid_t ut_pid;              /* Process ID of login process. */
    char ut_line[UT_LINESIZE]; /* Device name. */
    char ut_id[4];             /* Inittab ID. */
    char ut_user[UT_NAMESIZE]; /* Username. */
    char ut_host[UT_HOSTSIZE]; /* Hostname for remote login. */
    struct exit_status ut_exit; /* Exit status of a process marked
                                as DEAD_PROCESS. */

    /* The ut_session and ut_tv fields must be the same size when compiled
       32- and 64-bit. This allows data files and shared memory to be
       shared between 32- and 64-bit applications. */
    #if __WORDSIZE == 64 && defined __WORDSIZE_COMPAT32
        int32_t ut_session;     /* Session ID, used for windowing. */
        struct
        {
            int32_t tv_sec;      /* Seconds. */
            int32_t tv_usec;     /* Microseconds. */
        } ut_tv;               /* Time entry was made. */
    #else
        long int ut_session;     /* Session ID, used for windowing. */
        struct timeval ut_tv;    /* Time entry was made. */
    #endif

    int32_t ut_addr_v6[4];      /* Internet address of remote host. */
    char __unused[20];          /* Reserved for future use. */
};

```

这些文件给我们提供了所有用以构造`unpack()`语句的必需线索。`unpack()`取一个数据格式模板作为它的第一个参数。然后它使用该模板来决定如何对从第二个参数取得的二进制（通常是二进制）数据进行反汇编。`unpack()`将按格式拆分这些数据，返回一个列表，列表中的每个元素对应于所提供的模板中的相应元素。

基于Solaris *utmpx.h*包含文件中的C数据结构，让我们一步一步来构造我们的模板。有好些个模板字母是我们可以使用的。关于这些字符，我们会在表10-1中解释，但你还是应该查看*perlfunc*手册页中的`pack()`这一节来获取更多信息。构造模板有时候不是很直接简单的事，因为C编译器有时候会为了满足对齐的要求来填充数值。Perl自带的`pstruct`命令在面对这类问题的时候可以帮到我们。

表10-1：将utmpx.h的C代码转换为unpack()模板

C代码	unpack()模板	模板字母/重复数的说明
char ut_user[32];	A32	ASCII字符串，长度为32个字节（不足部分以空格填充）
char ut_id[4];	A4	ASCII字符串，长度为4个字节（不足部分以空格填充）
char ut_line[32];	A32	ASCII 字符串，长度为 32 个字节（不足部分以空格填充）
pid32_t ut_pid;	l	带符号的长整型值（4 个字节，和某些机器上真正的长整型大小可能不一样）
int16_t ut_type;	s	带符号的短整型值
struct {		
int16_t e_termination;	s	带符号的短整型值
int16_t e_exit;	s	带符号的短整型值
} ut_exit;		
	x2	编译器插入的空位填充
struct {	l	带符号的长整型值
time32_t tv_sec;		
int32_t tv_usec	l	带符号的长整型值
} ut_tv;		
int32_t ut_session;	l	带符号的长整型值
int32_t pad[5];	x20	跳过20个字节作为空位填充
int16_t ut_syslen;	s	带符号的短整型值
char ut_host[257];	Z257	ASCII字符串，以空字符串结尾，包含\0，长度为257个字节
	x	编译器插入的空位填充

模板构造好了，让我们把它用在真正的代码中：

```
# 针对 Solaris 9/10 wtmpx 的模板
my $template = 'A32 A4 A32 l s s s x2 l l l x20 s Z257 x';

my $recordsize = length( pack( $template, () ) );

open my $WTMP, '<', '/var/adm/wtmpx' or die "Unable to open wtmpx:$!\n";

my ($ut_user, $ut_id, $ut_line, $ut_pid,
    $ut_type, $ut_e_termination, $ut_e_exit, $tv_sec,
```



```

    $tv_usec, $ut_session,      $ut_syslen, $ut_host,
) = ();

# 一次读一行 wtmpx 记录
my $record;
while ( read( $WTMP, $record, $recordsize ) ) {

    # 用我们的模板 unpack 它
    (
        $ut_user, $ut_id,          $ut_line,  $ut_pid,
        $ut_type, $ut_e_termination, $ut_e_exit, $tv_sec,
        $tv_usec, $ut_session,      $ut_syslen, $ut_host
    ) = unpack( $template, $record );

    # 这段代码让输出更易读, 8 这个值取自  usr/include/utmp.h
    # 文件中:
    #   #define DEAD_PROCESS      8
    if ( $ut_type == 8 ) {
        $ut_host = '(exit)';
    }
    print "$ut_line:$ut_user:$ut_host:" . scalar localtime($tv_sec) . "\n";
}

close $WTMP;

```

下面是这段小程序的输出:

```

pts/176:vezt:c-61-212-209-21.hsd1.ma.comcast.net:Wed Apr 16 06:37:44 2008
pts/176:vezt:(exit):Wed Apr 16 06:38:03 2008
pts/147:birnou:pool-50-29-232-81.bos.eas.veriz.net:Wed Apr 16 08:09:27 2008
pts/17:croche:ce-23-213-189-154.nycap.res.rr.com:Wed Apr 16 08:34:18 2008
pts/17:croche:(exit):Wed Apr 16 08:34:45 2008
pts/139:hermd:d-66-249-250-270.hsd1.ut.comc.net:Wed Apr 16 09:45:57 2008
pts/139:hermd:(exit):Wed Apr 16 09:58:55 2008

```

在继续之前, 上面这段代码中的有些地方需要说明一下: `read()` 函数的第三个参数是它将读取的字节数。相对于将要读取的记录大小 (比如说 “32” 个字节) 写死在代码里, 我们更倾向于使用 `pack()` 函数的一个方便的属性。当传过来的是个空列表, `pack()` 函数会返回一个以空值或是空格填充的字符串, 大小和记录本身的大小相同。这样我们就可以给 `pack()` 函数指定任意的记录模板, 然后让它自己来告诉我们该模板对应的记录大小是多少:

```
my $recordsize = length( pack( $template, () ) );
```

`unpack()` 方法并不是访问 `wtmp/x` 数据唯一的 Perl 内置方法。至少有一个模块使用了被系统提供商所认可的系统调用 (`getutxent()` 等) 来读取这些文件。我们稍后将在一个范例中使用这个模块。

用户会因此而感到强大

所有我们介绍过的访问日志信息的方法中，`unpack()`方法最有可能让你感觉自己作为用户拥有强大的力量。当其他方法因为数据本身的损坏而失效时，就是这个方法展现用途的时候了。举例说来，我听说过这样一个案例，`wtmpx`文件损坏，如果用`last`命令查看的话会不断跳出错信息。当这种情况发生，有时候我们是可以写些代码跳过坏掉的部分内容（通过使用`sysread()`函数配合`unpack()`），从而允许我们恢复这个文件的其他好的部分。当你在这种情形中使用它解决了问题，肯定会让你有英雄般的感觉。

调用操作系统（或其他）二进制文件

由于审查`wtmp`文件是很普遍的任务，所以Unix系统都附带了名为`last`的命令来以可读的形式打印出这个二进制文件的内容。下面的输出样例和前面我们所举的范例中的输出几乎是一样的：

```
vezt      pts/176      c-61-212-209-21. Wed Apr 16 06:37 - 06:38 (00:00)
birnou    pts/147      pool-50-29-232-8 Wed Apr 16 08:09  still logged
croche    pts/17       ce-23-213-189-15 Wed Apr 16 08:34 - 08:34 (00:00)
hermd     pts/139      d-66-249-250-270 Wed Apr 16 09:45 - 09:58 (00:12)
```

我们可以很容易地在Perl中调用如`last`这样的二进制文件。下面的代码将不重复地显示所有在当前`wtmpx`文件中找到的用户名：

```
# last 命令二进制文件的路径
my $lastexec = '/bin/last';

open my $LAST, '-|', "$lastexec" or die "Unable to run $lastexec:$!\n";
my %seen;
while(my $line = <$LAST>){
    last if $line =~ /^$/;
    my $user = (split(' ', $line))[0];
    print "$user\n" unless exists $seen{$user};
    $seen{$user}='';
}
close $LAST or die "Unable to properly close pipe:$!\n";
```

既然`unpack()`可以满足我们所有的要求，那我们为什么还要使用上面提到的这种方法呢？原因是可移植性。如你所见，`wtmp/x`文件的格式在不同的Unix变种间可能是有差异的。在此之上，某个系统提供商还可能在不同的系统版本间造成`wtmp/x`文件格式的不一致，这会直接导致你之前完美的`unpack()`模板失效。

尽管如此，你能依靠的便是一直存在的可以阅读这个格式文件的`last`命令，使用它你就

可以和底层格式改变相对独立，不受影响。如果你使用`unpack()`方法，那么针对需要解析的不同格式的`wtmp`文件，你不得不创建并管理多个单独的模板字符串^[注2]。

相比于`unpack()`，使用这个方法的最大缺点就是在程序中解析所需字段的复杂程度增加了。使用`unpack()`，所有的字段都会自动地从数据中给你抽取出来。使用我们的`last`范例，你可以找到对`split()`函数或者正则表达式不友好的输出行（如下），这些输出行都混在一起：

```
user   console                               Wed Oct 14 20:35 - 20:37 (00:01)
user   pts/12      208.243.191.21  Wed Oct 14 09:19 - 18:12 (08:53)
user   pts/17      208.243.191.21  Tue Oct 13 13:36 - 17:09 (03:33)
reboot system boot                Tue Oct 6 14:13
```

我们的眼睛要区分输出里面的每一列可能会比较困难，但是任何用来解析这个输出的程序都得处理第一行和第四行中信息空白的部分。`unpack()`在这种情况下仍然可以正确地解析内容，因为它是按固定长度的字段来解析的。尽管如此，它也不是任何时候都管用。关于写更高级的解析器，有另外的技术，但对我们而言可能太复杂了。

使用操作系统的日志记录API

为了介绍这个方法，我们需要将重点转到Windows事件日志服务。之前提到过，很不幸地，Windows机器不会将日志记录到纯文本文件。要访问日志文件数据，唯一支持的方法是通过调用一系列API（应用程序编程接口）。很多用户依赖于如图10-1所示的事件查看器程序来获取他们想要的数据库。

幸运的是，我们有个Perl模块（由Jesse Dougherty编写，之后由Martin Pauley和Bret Giddings更新）允许我们轻松地访问事件日志API调用^[注3]。下面这个简单的程序会以类似`syslog`的格式输出`System`事件日志中的事件列表，在本章的稍后部分我们会介绍该程序的更复杂版本：

```
use Win32::EventLog;
# 每个事件都有个类型——下面是常见事件类型的说明
my %type = (1 => 'ERROR',
            2 => 'WARNING',
            4 => 'INFORMATION',
```

注2：这种方法也有点不足，因为你仍然需要去跟踪`last`可执行程序在各个Unix环境中的路径；另外，对程序的输出在格式上的差异你也得去处理。

注3：也可以使用之前在第4章中接触过的Windows管理规范（WMI）框架来获取Windows中的日志信息，但是相比而言`Win32::EventLog`更易于使用和理解。如果你需要解析存储在非Windows机器上的Windows事件日志数据，可以试试John Eaglesham的`Parse::EventLog`模块。

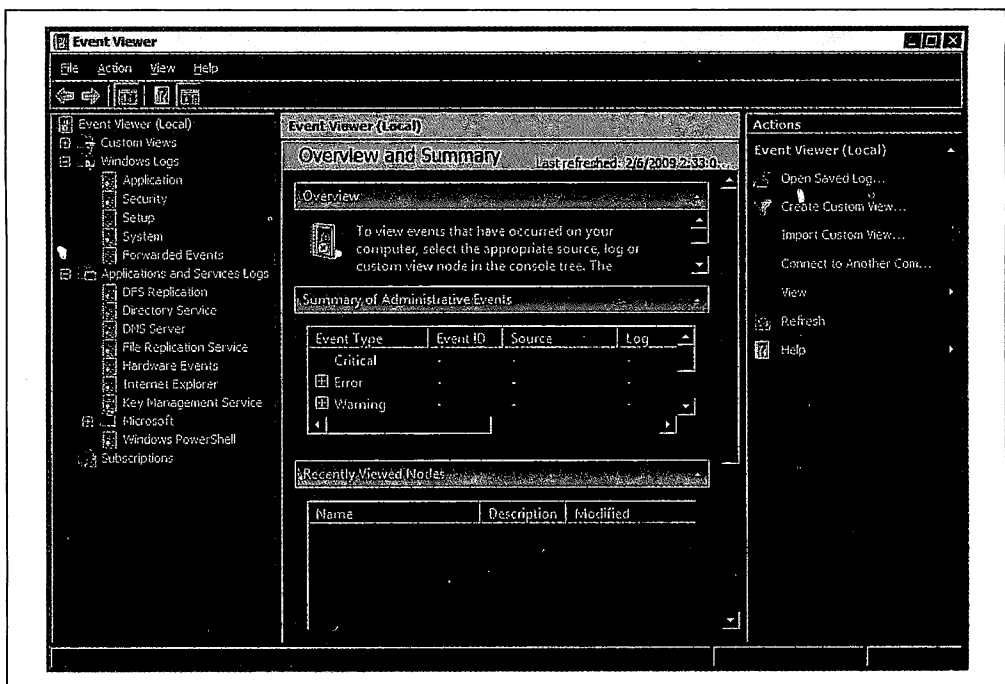


图10-1: Windows事件查看器

```

8 => 'AUDIT_SUCCESS',
    16 => 'AUDIT_FAILURE');

# 假如设置了这个选项，在每次调用Read()的时候我们都将读取
# 每个消息的全文
$Win32::EventLog::GetMessageText = 1;

# 打开 System 事件日志
my $log = new Win32::EventLog('System')
    or die "Unable to open system log:$^E\n";

my $event = '';
# 从第一条开始，每次读取一个记录
while ($log->Read((EVENTLOG_SEQUENTIAL_READ|EVENTLOG_FORWARDS_READ),
    1,$entry)){
    print scalar localtime($entry->{TimeGenerated}).' ';
    print $entry->{Computer}.'['.$entry->{EventID} &
        0xffff).'] ';
    print $entry->{Source}.':'.$type{$entry->{EventType}}.':';
    print $entry->{Message};
}

```

在Windows上也有类似于`last`命令那样将事件日志输出为纯ASCII格式的工具。在本章后面部分我们会介绍其中一个这样的工具。另外，接下来我们还将介绍一个和Unix操作系统处理`wtmp`数据相同的，使用类似的操作系统日志记录API的例子。

日志文件数据结构

除了日志文件的数据格式外，了解这些文件本身的内容也非常重要，因为数据表示的是什么以及数据是如何来表示的，这两者都是我们在编程解决问题时需要考虑的问题。就日志文件内容来说，我们常常可以注意到这样两个特征：有状态（stateful）数据和无状态（stateless）数据。让我们通过两个例子来清楚认识一下这两个特征。

下面是取自Apache Web服务器日志的三行片段。每一行都代表了服务器答复的一条请求：

```
esnet-118.dynamic.rpi.edu - - [13/Dec/2008:00:04:20 ?0500] "GET home/u1/tux/tuxed05.gif
HTTP/1.0" 200 18666 ppp-206-170-3-49.okld03.pacbell.net - - [13/Dec/2008:00:04:21
?0500] "GET home/u2/news.htm
HTTP/1.0" 200 6748 ts007d39.ftl-fl.concentric.net - - [13/Dec/2008:00:04:22 ?0500]
"GET home/u1/bgc.jpg HTTP/1.1" 304 -
```

下面是取自打印机守护进程日志文件的几行记录：

```
Aug 14 12:58:46 warhol printer: cover/door open
Aug 14 12:58:58 warhol printer: error cleared
Aug 14 17:16:26 warhol printer: offline or intervention needed
Aug 14 17:16:43 warhol printer: error cleared
Aug 15 20:40:45 warhol printer: paper out
Aug 15 20:40:48 warhol printer: error cleared
```

在这两个例子中，日志文件中的每一行都和文件中的其他行是相对独立的，它们之间没有什么关系。虽然我们可以从中找到一定的模式或将某些行合在一起收集统计信息，但是日志文件的每条记录数据之间并没直接的内在联系。

好的，现在让我们看看几条稍微修改过的sendmail邮件日志记录：

```
Dec 13 05:28:27 mailhub sendmail[26690]: FAA26690:
from=<user@has.a.godcomplex.com>, size=643, class=0, pri=30643, nrcpts=1,
msgid=<200812131032.CAA22824@has.a.godcomplex.com>, proto=ESMTP,
relay=user@has.a.godcomplex.com [216.32.32.176]

Dec 13 05:29:13 mailhub sendmail[26695]: FAA26695: from=<root@host.example.edu>,
size=9600, class=0, pri=39600, nrcpts=1,
msgid=<200812131029.FAA15005@host.example.edu>, proto=ESMTP,
relay=root@host.example.edu [192.168.16.69]

Dec 13 05:29:15 mailhub sendmail[26691]: FAA26690: to=<user@host.example.edu>,
delay=00:00:02, xdelay=00:00:01, mailer=local, stat=Sent

Dec 13 05:29:19 mailhub sendmail[26696]: FAA26695: to="|IFS=' '&&exec /usr/bin/
```

```
procmail -f-||exit 75 #user", ctladdr=user (6603/104), delay=00:00:06,  
xdelay=00:00:06, mailer=prog, stat=Sent
```

和之前的例子不同，这个文件中的各行内容之间有明显的联系。图10-2中的内容会让你更清楚地了解它们之间的联系。

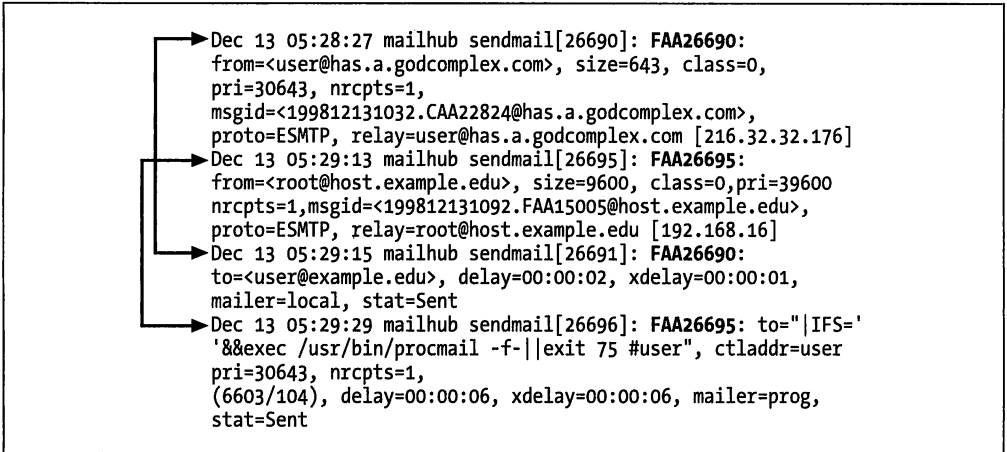


图10-2: sendmail日志中的相关记录

每行都至少有一个相应的记录来显示各条消息的来源地和目的地。当一个消息进入到系统的时候，会被分配一个唯一的“Message-ID”，如图中高亮部分所示，它标识出正在处理的是哪条消息。这个消息ID允许我们在交错记录的日志文件中将相关的记录关联起来，本质上就是赋予了一条消息在日志文件各条记录间的存在性或“状态”。

有时候我们关心状态转换之间的“间隔”。用本章之前介绍过的*wtmpx*文件来举例：在该文件中，我们感兴趣的不仅仅只是什么时候用户登录和注销（这就是日志中的两种状态转换），我们还想知道发生这两个事件之间的时间间隔（也就是这个用户登录进来多长时间）。

还有更复杂的日志文件，处理起来更难。下面是从处于调试模式的POP（Post Office Protocol，邮局协议）服务器日志文件中摘录的一小段。为了保护用户隐私，其中的用户名和IP地址是被修改过的。

```
Jan 14 15:53:45 mailhub popper[20243]: Debugging turned on
Jan 14 15:53:45 mailhub popper[20243]: (v2.53) Servicing request from
"client" at 129.X.X.X
Jan 14 15:53:45 mailhub popper[20243]: +OK QPOP (version 2.53) at mailhub starting.
Jan 14 15:53:45 mailhub popper[20243]: Received: "USER username"
Jan 14 15:53:45 mailhub popper[20243]: +OK Password required for username.
Jan 14 15:53:45 mailhub popper[20243]: Received: "pass xxxxxxxxx"
Jan 14 15:53:45 mailhub popper[20243]: +OK username has 1 message (26627 octets).
```

```
Jan 14 15:53:46 mailhub popper[20243]: Received: "LIST"
Jan 14 15:53:46 mailhub popper[20243]: +OK 1 messages (26627 octets)
Jan 14 15:53:46 mailhub popper[20243]: Received: "RETR 1"
Jan 14 15:53:46 mailhub popper[20243]: +OK 26627 octets
<此处为消息文本>
Jan 14 15:53:56 mailhub popper[20243]: Received: "DELE 1"
Jan 14 15:53:56 mailhub popper[20243]: Deleting message 1 at offset 0 of length 26627
Jan 14 15:53:56 mailhub popper[20243]: +OK Message 1 has been deleted.
Jan 14 15:53:56 mailhub popper[20243]: Received: "QUIT"
Jan 14 15:53:56 mailhub popper[20243]: +OK Pop server at mailhub signing off.
Jan 14 15:53:56 mailhub popper[20243]: (v2.53) Ending request
from "user" at (client) 129.X.X.X
```

这里我们不但有连接（“Servicing request from……”这行）和断开连接（“Ending request from……”这行）的信息，还有在这两个状态转换之间的详细信息。

其中的每个事件也给我们提供了潜在的有用的状态转变之间的“间隔”信息。如果我们的POP服务器出问题了，我们可能需要了解日志输出中的每一步花了多长时间。

在FTP服务器上，你可以从类似这样的数据中总结出用户和你的站点之间的交互情况。通常，在用户建立连接后，平均得花多长时间才开始传输文件？他们在发送每个指令间会暂停很长时间么？他们在下载相同的文件前会先在你的站点上四处溜达么？这些存在时间间隔的数据可以给你提供丰富的信息。

处理日志文件信息

一旦你学会了如何通过编程方式访问日志信息，那么接下来你将要面对的两件应用层面比较重要的事情就浮现了：日志信息空间管理和日志分析。接下来我们将逐一介绍它们。

日志信息的空间管理

让程序提供有用或详尽的日志输出的负面影响是它会占用大量的磁盘空间。在本书所覆盖的三大操作系统中（Unix、Mac OS X和Windows），这个都是值得考量的问题。Windows可能是其中最不需要关心这个问题的操作系统，因为它的集中化日志工具有内置的功能支持日志自动剪裁。

通常，将日志文件保持在一个合理大小的任务是由系统管理员来承担的。大多数的Unix变种都随系统提供了一系列自动化的日志大小管理机制，但往往它只处理系统自带的一些日志文件。当你给机器添加了另外一种服务并且它会产生独立的日志文件，此时对系统发行商所提供的解决方案进行调整就非常有必要了。

日志轮循

解决空间占用问题的常用方法是对日志文件进行轮循（rotate）。（下一小节我们将探究一种不常见的解决方法）。当过了指定的时间间隔或者文件达到了某个特定大小时，我们就将当前日志文件重命名（比如将logfile重命名为logfile.0）。然后日志记录进程会继续向新的空文件里面输出日志。下一次过了指定的时间间隔或文件大小超过限定值时，我们便重复上面的操作，首先会将之前备份的文件重命名（如将logfile.0重命名为logfile.1），然后将当前日志文件重命名为logfile.0。这个过程会像这样一直重复，直到创建了一定数量的日志备份文件，此时最老的日志备份文件就会被删除。图10-3描述了这个过程。

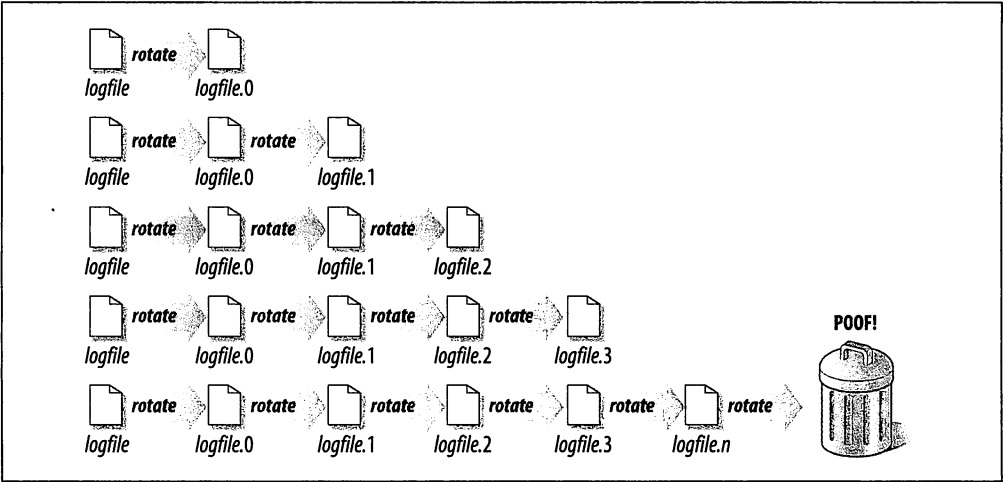


图10-3：日志轮循的图形表示

通过这个方法，我们便可以保留合理的、限量的日志数据。表10-2提供了日志轮循的方法以及实现每一步所需的Perl函数。

表10-2：使用Perl实现日志轮循的方法

过程

将老的日志备份移走（即分别将它们按序重命名或移动到新文件）

如果有必要，给创建该日志文件的进程发送信号，通知其关闭当前文件并停止记录日志到磁盘，直到接到新通知为止

Perl

使用rename()函数，或者在需要跨文件系统移动文件时使用File::Copy::move()模块

针对可以接受信号的程序，可以用kill()函数；如果需要调用其他管理程序，则可以使用system()或``（反引号）

表10-2: 使用Perl实现日志轮循的方法 (续)

过程	Perl
复制或移动刚才还在使用中的日志文件到另一个文件	可以使用File::Copy模块来复制文件, 用rename()函数来重命名 (如果需要跨文件系统移动文件, 可以使用File::Copy::move())
如果需要, 清空当前的日志文件	使用truncate()函数或者open my \$FILE,'>', 'filename'
如果需要, 发送信号告诉日志进程重新恢复日志记录	见本表第2步
如果需要, 对复制好的文件进行压缩或进一步处理	可以使用system()函数或者`` (反引号) 来运行压缩程序; 使用; Compress::Zlib模块或者其他代码来做进一步处理
删除其他老的日志文件复件	使用stat()函数来检查文件大小及日期; 用unlink()函数来删除文件

有很多各不相同的内容是关于这个主题的。很多人都自己编写过脚本来处理日志轮循。因此, Perl有一个模块来专门处理日志轮循也就不足为奇了。我们将要介绍一下Paul Gampe的Logfile::Rotate模块。

Logfile::Rotate模块使用面向对象的编程惯例, 需要创建一个日志文件对象实例, 然后调用该实例的方法。首先, 让我们创建一个新的实例, 参数如表10-3所示。

表10-3: Logfile::Rotate的参数

参数	目的
File	需要轮循的日志文件的名称
Count (可选参数, 默认值: 7)	需要保留的备份文件数目
Gzip (可选参数, 默认值: Perl在编译时找到的默认的gzip程序, 应该在你的当前可执行文件路径中)	gzip压缩程序的完整路径
Post	轮循完成后将要执行的代码, 和表10-2中的第5步相同

下面是一些使用这些参数的代码范例:

```
use Logfile::Rotate;
my $logfile = new Logfile::Rotate(
    File    => '/var/adm/log/syslog',
    Count   => 5,
```

```

Gzip  => '/usr/local/bin/gzip',
Post  =>
    sub {
        open my $PID, '<', '/etc/syslog.pid' or
            die "Unable to open pid file:!\n";
        chomp(my $pid = <$PID>);
        close $PID;
        kill 'HUP', $pid;
    }
);
# 日志文件被锁定并被加载。现在开始进行轮循。
$logfile->rotate();
# 确保日志文件被解锁（销毁对象会将文件解锁）
undef $logfile;

```

注意：前面的这段代码有三个潜在的安全缺陷。看看你是否能在查看以下补充内容中关于如何避免这三个安全缺陷的答案及技巧前自己发现它们。

找到并修复不安全的代码

你已经仔细阅读了这段Logfile::Rotate代码来寻找安全漏洞，那我们就来谈谈这个吧。由于该模块经常是以特权用户的身份（如root用户）来运行，因此有几点需要考虑：

- /usr/local/bin/gzip命令会以相同的特权用户身份运行。其中我们通过完整路径来调用外部命令是很正确的（很重要！），但是你理应检查一下谁具有在文件系统上对该可执行文件的修改/替换权限。另外一个更安全的绕过这个问题（假定你能完全控制谁能安装Perl模块）的方法是改用Gzip => 'lib'。这样Logfile::Rotate模块会调用Compress::Zlib模块代替调用独立的二进制文件来做压缩工作。
- 在Post环节，我们的代码会很高兴地读取/etc/syslog.pid文件，并不会在意这个文件是否会被恶意分子篡改。该文件是否是所有人都可写入的？它是否指向其他文件？该文件的拥有者是否正确？这些我们的代码都不关心，但事实上它应该关心。在执行操作前，应该通过stat()函数检查文件的属性。
- 在相同的环节中，我们的代码轻率地对从文件（刚提到过）中读取到的PID发送了HUP信号。它没有去检测那个进程ID实际上是不是指向一个正在运行的syslog进程。如果更小心一点编码，应该在发送信号前检查进程表（或许采用在第4章中讨论过的指定进程表的策略）。

以上这些是这段代码最主要的问题。针对这些问题，请务必阅读第1章中脚本安全的相关章节。

循环缓冲

我们刚刚讨论了以传统的日志轮循的方法来处理日益增长的日志带来的存储问题。现在让我向你介绍另外一种不一样的处理该问题的方式。

这里我们举一个常见的场景：你正试图去调试某个日志输出量巨大的服务器守护进程。但其中你感兴趣的只是日志输出总量中的一小部分，也许只是当你用特定的客户端运行一些测试后服务器所产生的那几行日志。如果像往常一样将所有的日志输出都保存到磁盘，磁盘很快会被塞满。而在这种输出量的情况下勤快地进行日志轮循会拖慢服务器。此时你该怎么办？

我写了个名为*bigbuffy*的程序来解决这个难题。使用方式比较直接。*bigbuffy*每次从“标准”或“控制台”输入读取一行，这些行被存储在指定大小的循环缓冲区中（参见图10-4）。每当缓冲区被占满，就从缓冲区头部继续覆盖记录。这个读取/存储的过程一直进行，直到*bigbuffy*收到来自用户的信号。收到信号后，它会将缓冲区当前的内容转储到一个文件，然后返回继续做它平常该做的事。此时磁盘上截下的就是日志流的其中一段，包含着你所需要的数据。

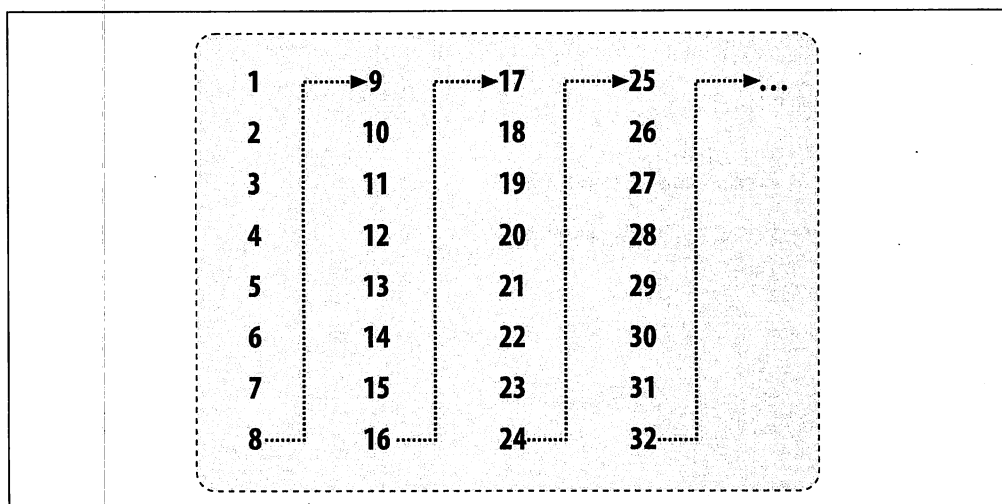


图10-4：将日志记录到循环缓冲区

*bigbuffy*可以和第13章中会提到的类似的服务监控程序搭配工作。一旦监控器发现了问题，它就可以发送信号给*bigbuffy*，通知它进行日志缓冲区的转储，这样便能给你留下一份发生问题时的日志快照（假定你的缓冲区够大，且监控器发现问题够及时）。

下面有个简化版的*bigbuffy*。代码的长度是本章我们所见范例中最长的，但它并不复杂。稍后我们将把它作为一个跳板来定位一些很重要的问题，如输入被阻塞或安全问题：

```

use Getopt::Long;

my @buffer;          # 用以保存输入的缓冲区
my $dbuffsize = 200; # 默认的循环缓冲区大小（以行计）
my $whatline = 0;    # 循环缓冲区的开始行号
my $dumpnow = 0;     # 用以指示转储请求的标志

# 解析命令行选项
my ( $buffsize, $dumpfile );
GetOptions(
    'buffsize=i' => \$buffsize,
    'dumpfile=s' => \$dumpfile,
);
$buffsize ||= $dbuffsize;

# 设置信号处理器并初始化计数器
die "USAGE: $0 [--buffsize=<lines>] --dumpfile=<filename>"
    unless ( length($dumpfile) );

$SIG{'USR1'} = \&dumpnow; # 为转储设置好信号处理器

# 开始干活儿！（简单的“行读取/存储”循环）
while ( defined( $_ = <> ) ) {

    # 将行插入到数据结构中。
    # 注意：这是我们首先要做的事情，哪怕我们捕获了一个信号。
    # 宁愿多转储一行内容，以免转储过程中出现问题导致漏掉一行数据

    $buffer[$whatline] = $_;

    # 下一行到缓冲区的什么位置？
    $whatline = ++$whatline % $buffsize;

    # 如果收到信号，就进行当前缓冲区的转储
    if ( $dumpnow ) {
        dodump();
    }
}

# 简单的信号处理器，仅用来设置异常标志
# 参见手册 perlipc(1)
sub dumpnow {
    $dumpnow = 1;
}

# 将循环缓冲区转储至一个文件，如果该文件已存在，则追加到这个文件中
sub dodump {
    my $line;          # 统计被转储的行
    my $exists;        # 标志，输出文件是否已经存在？
    my $DUMP_FH;       # 转储文件的文件句柄
    my ( @firststat, @secondstat ); # 用来保存 lstats 的输出

    $dumpnow = 0;      # 重置该标志及信号处理器
    $SIG{'USR1'} = \&dumpnow;

```

```

if ( -e $dumpfile and ( ! -f $dumpfile or -l $dumpfile ) ) {
    warn 'ALERT: dumpfile exists and is not a plain file, '.
        "skipping dump.\n";
    return undef;
}

# 当进行文件追加操作的时候，我们需要有一些特殊的预防措施。
# 下面的一系列 “if” 语句会在打开文件进行追加操作前
# 先进行一系列安全性测试。
if ( -e $dumpfile ) {
    $exists = 1;
    unless ( @firststat = lstat $dumpfile ) {
        warn "Unable to lstat $dumpfile, skipping dump.\n";
        return undef;
    }
    if ( $firststat[3] != 1 ) {
        warn "$dumpfile is a hard link, skipping dump.\n";
        return undef;
    }
}
unless ( open $DUMP_FH, '>>', $dumpfile ) {
    warn "Unable to open $dumpfile for append, skipping dump:!.\\n";
    return undef;
}
if ($exists) {
    unless ( @secondstat = lstat $DUMP_FH ) {
        warn "Unable to lstat opened $dumpfile, skipping dump.\n";
        return undef;
    }

    if (
        $firststat[0] != $secondstat[0] or    # 检查设备号
        $firststat[1] != $secondstat[1] or    # 检查 inode
        $firststat[7] != $secondstat[7]       # 检查文件大小
    ) {
        warn "SECURITY PROBLEM: lstats don't match, skipping dump.\n";
        return undef;
    }
}

$line = $whatline;
print {$DUMP_FH} '-' . scalar(localtime) . ( '-' x 50 ) . "\\n";

do {
    # 若缓冲区并没有被填满，则只打印出有效的行
    print {$DUMP_FH} $buffer[$line] if defined $buffer[$line];
    $line = ++$line % $buffsize;
} until $line == $whatline;

close $DUMP_FH;

# 清空当前缓冲区以避免残留内容被用在之后的转储过程中
$whatline = 1;

```

```
@buffer = ();  
  
return 1;  
}
```

类似这样的程序会带来一些有趣的实现上的问题。接下来我们将要介绍其中的一些。

日志处理程序的输入阻塞问题。之前提到过，这是一个简化版的*bigbuffy*。考虑的是实现时的简单性，尤其是跨平台性，因此这个版本的程序有一些缺陷：当将数据转储到磁盘的时候，它没法同时继续接受输入。所以在缓冲区进行转储的时候，操作系统可能会通知给*bigbuffy*发送数据的程序先暂停操作，等待并耗尽它自己的输出缓冲。不过幸运的是，转储过程很快速，所以发生这种事情的时间是很短暂的，但无论如何，我们可能对这点缺陷不会很喜欢。

解决这个问题可能的方法有：

- 使用双重缓冲和多任务处理的方式重写*bigbuffy*。不同于使用单个存储缓冲区，这个解决方法会用两个缓冲区。当接收到信号的时候，程序会开始把日志记录到第二个缓冲区，而使用一个子进程或其他的线程来处理第一个缓冲区的转储操作。当下一个信号到来的时候，这两个缓冲区的角色进行互换，如此反复。
- 或者也可以重写*bigbuffy*，让它在转储的时候使用交错读写的方式来进行处理。这个方式最简单的一个实现可能是这样的：它需要我们在每次读取一行新内容时，就写几行记录到输出文件。但当日志不是稳定流式地输出，而是一段一段迸发式地输出时，情况会变得比较复杂一点。我们可以用类似超时或定时的机制来解决这样的问题。

由于这两种解决方法都很难回避跨平台的问题，所以我们在本书中给出的是简化版本的程序范例。

日志处理程序的安全性。你也许注意到了，*bigbuffy*对输出文件的打开和写操作都有相当程度的安全性考虑。这也是在之前“日志轮循”一节里面提到的防患于未然型的编码方式的一个例子。如果这个程序被用来调试服务器守护进程，它很有可能是以特权用户身份在系统上运行的。因此，将有可能使程序被滥用的异常情形考虑进去是十分重要的。

一个可能的场景是指向输出文件的链接被指向另外一个文件的链接替换掉了。如果我们在打开和写文件之前没有检查该文件的身份，很可能无意间把很重要的文件，如*/etc/passwd*，给破坏掉了。甚至就算我们在打开文件前检查了该输出文件，还是有可能在我们对它进行写操作之前被心怀不轨的家伙换成其他文件。为了避免这种情况，我们可以：

- 检查输出文件是否已经存在。如果是，我们就用`lstat()`函数来获取与它相关的文件系统信息。
- 我们以追加模式打开文件。
- 在实际对该文件进行写操作前，我们对打开的文件句柄使用`lstat()`函数，用来检查它是否仍然是我们期望操作的那个文件，是否自最初我们检查过后那个文件没有被替换过。如果发现不是同一个文件（比如说，有人在`open`前把该文件替换成了其他链接），我们就不对该文件进行写操作并大声警告。最后的这一步也避免了在第1章中提到过的潜在竞争条件。

如果我们不需要对文件进行追加，大可以改为打开一个随机名字（这样文件名就不会预先被猜测到）的临时文件，然后再把该临时文件重命名为我们期望的文件。Perl发行版自带了Tim Jenness的`File::Temp`模块来帮助我们完成这样的工作。

这类周旋的手段在大多数的Unix系统上是必要的，因为Unix系统最初并不是把安全问题作为高优先级来设计的。Windows也有类似的“junction”（联接）^[注4]，它基本上等同于符号链接，但目前还没有任何迹象表明由于它的实现而存在类似的安全威胁。

日志解析及分析

有些系统管理员觉得日志的轮循就是他们需要关心的唯一事情。哪怕调试时所需要的信息都已经在磁盘上，他们都不会对这些数据有任何其他加以利用的念头。我想说的是，这种想法是非常短视的，哪怕一点点的日志文件分析都可能给我们带来很大的帮助。接下来我们将介绍一些使用Perl进行日志分析的方法，我们将从简单的例子开始，慢慢由浅入深。

这节中大部分的例子都是用Unix的日志文件来做演示的，因为通常的Unix系统都有大量的日志文件等着被分析，尽管如此，这里介绍的方法并不只是针对特定操作系统的。

流式读取－统计

最简单的方法是“读取并统计”，也就是我们读取日志流并从中寻找感兴趣的数据，一旦找到相关的数据，就累加计数器。下面这个简单的例子会从Solaris 10的`wtmpx`文件内容中统计出机器重启的次数：

注4： 你可能还听过“reparse point”（重解析点）这个专业术语，但微软在发行了一些系统版本之后对它进行了修订。在本书写作之时，“junction”被认为是创建自“reparse point”。

```

# Solaris 10 wtmpx 文件的模板
my $template = 'A32 A4 A32 l s s x2 l l l x20 s Z257 x';

# 检查记录的大小
my $recordsize = length( pack( $template, ( ) ) );

# 打开文件
open my $WTMP, '<', '/var/adm/wtmpx' or die "Unable to open wtmpx:!\n";

my ( $ut_user, $ut_id,          $ut_line,  $ut_pid,
     $ut_type, $ut_e_termination, $ut_e_exit, $tv_sec,
     $tv_usec, $ut_session_pad,  $ut_syslen, $ut_host
   )
  = ( );

my $reboots = 0;

# 一次一行地从中读取数据
while ( read( $WTMP, $record, $recordsize ) ) {
    (  $ut_user, $ut_id,          $ut_line,  $ut_pid,
      $ut_type, $ut_e_termination, $ut_e_exit, $tv_sec,
      $tv_usec, $ut_session,      $ut_syslen, $ut_host
    )
    = unpack( $template, $record );

    if ( $ut_line eq 'system boot' ) {
        print "rebooted " . scalar localtime($tv_sec) . "\n";
        $reboots++;
    }
}

close $WTMP;
print "Total reboots: $reboots\n";

```

接下来，让我们对这个方法论做一些扩展，研究一下使用Windows事件日志工具来进行信息统计的一个例子。之前提过，Windows有设计良好和较复杂的系统日志记录机制。这种复杂性对Perl程序员新手来说有些困难。我们将使用一些Windows平台特定的Perl模块例程来获取基本的日志信息。

Windows程序和操作系统组件会通过发送“事件”给不同的事件日志来记录它们的活动。系统会记录这些基本的日志信息，如事件什么时候被发送过来、哪个程序或操作系统函数发送的该事件、该事件是什么类型的（只是简单的介绍性信息或是其他更紧要的信息）等。

和Unix不一样，事件的真实描述或日志消息实际上并没有和事件记录存放在一起。换言之，在日志记录中它提供的是一个EventID。这个EventID包含一个对编译至一个程序库（即.dll文件）中的特定消息的引用。由给定的EventID来取得相应的日志信息是不容易的。这个过程涉及从注册表中查找出相应的库并手动加载这些库。幸运的是，当前版

本的Win32::EventLog模块能帮我们自动完成这项工作（参见“使用操作系统的日志记录API”一节中Win32::Eventlog模块的第一个例子中的\$Win32::EventLog::GetMessageText）。

在接下来的例子中，我们将基于当前System日志中的事件记录来生成一些统计信息，包括它们从哪里来、它们的重要性。这个程序采用了与本章第一个Windows日志记录范例不同的方式。

首先我们加载Win32::EventLog模块，它黏合了Perl和Windows事件日志记录例程。接着，我们初始化一个哈希表来保存调用日志读取例程所产生的结果。一般情况下Perl会帮我们处理好这个过程，但有时候我们自己加上这段代码有益于其他需要阅读该程序的人。最后，我们建立一个小的事件类型列表，这些列表在之后打印统计信息时会被用到：

```
use Win32::EventLog;

# 这里和 $event{Length => NULL, RecordNumber =>NULL, ...} 等效
my %event;
my @fields = qw(Length RecordNumber TimeGenerated TimeWritten EventID
    EventType Category ClosingRecordNumber Source Computer Strings Data);
@event{@fields} = (NULL) x @fields;

# 部分事件类型的列表：类型 1 代表 “Error”，
# 2 代表 “Warning” 等。
my @types = ('','Error','Warning','','Information');
```

下一步我们要打开System事件日志。通过调用Open()函数，我们可以获得一个EventLog句柄并将其保存在\$EventLog变量中，它被用来作为我们和这个特定日志之间的连接：

```
my $EventLog = ''; # 事件日志句柄
my $event = ''; # 我们将要返回的事件
my $numevents = 0; # 日志中事件的总数
my $oldestevent = 0; # 日志中最旧的事件
Win32::EventLog::Open($EventLog,'System','')
    or die "Could not open System log:$^E\n";
```

当我们得到该句柄后，我们可以使用它来获取日志中事件的数量和最旧记录的ID：

```
$EventLog->GetNumber($numevents);
$EventLog->GetOldest($oldestevent);
```

我们将这些信息作为第一个Read()语句的一部分，它可以帮我们定位到日志的第一条记录。这和用seek()函数来定位到文件开头是一样的：

```
$EventLog->Read( ( EVENTLOG_SEEK_READ | EVENTLOG_FORWARDS_READ ),
    $numevents + $oldestevent, $event );
```

从这儿开始，我们使用一个简单的循环来分别读取每一条日志记录。EVENTLOG_SEQUENTIAL_READ标志指“从上一个记录读取的位置继续读取”。EVENTLOG_FORWARDS_READ标志能让我们按时间顺序进行处理^[注5]。Read()的第三个参数是记录的偏移量：在本例中为0，因为我们想从上次离开的地方继续读取。当我们读完每一条记录后，代码会将它们的Source和EventType属性记录在作为计数器的哈希表中。

```
my %source;
my %types;
for ( my $i = 0; $i < $numevents; $i++ ) {
    $EventLog->Read( ( EVENTLOG_SEQUENTIAL_READ | EVENTLOG_FORWARDS_READ ),
        0, $event );
    $source{ $event->{Source} }++;
    $types{ $event->{EventType} }++;
}

# 打印出总数
print "--> Event Log Source Totals:\n";
for ( sort keys %source ) {
    print "$_: $source{$_}\n";
}
print '-' x 30, "\n";
print "--> Event Log Type Totals:\n";
for ( sort keys %types ) {
    print "$types{$_}: $types{$_}\n";
}
print '-' x 30, "\n";
print "Total number of events: $numevents\n";
```

我运行的结果看起来像这样：

```
--> Event Log Source Totals:
Application Popup: 4
BROWSER: 228
DCOM: 12
Dhcp: 12
EventLog: 351
Mouclass: 6
NWCWorkstation: 2
Print: 27
Rdr: 12
RemoteAccess: 108
SNMP: 350
Serial: 175
Service Control Manager: 248
Sparrow: 5
Srv: 201
msbusmou: 162
msi8042: 3
msinport: 162
```

注5： 这是 Win32 事件日志记录例程很灵活的另一个体现。通过这种功能，在必要的时候，我们的代码就可以做到先移动到日志的末尾，然后按时间来倒序读取日志记录。

```

mssermou: 151
qic117: 2
-----
--> Event Log Type Totals:
Error: 493
Warning: 714
Information: 1014
-----
Total number of events: 2220

```

由于之前我们承诺过，所以这里给出一个范例代码，它依赖于类似*last*的程序来转储事件日志的内容。这个程序使用的是Jesper Lauritsen的*ElDump*程序，你可以从<http://www.ibt.ku.dk/Jesper/NTtools/>下载。*ElDump*和*DumpEl*程序功能相似，它可以在很多源码套件中获得（也可以在<http://www.microsoft.com/windows2000/techinfo/reskit/tools/existing/dumpel-o.asp>在线获得）：

```

my $eldump = 'c:\bin\eldump';    # ElDump 的路径

# 输出数据字段之间以 ~ 符号分隔，不包含完整消息
# 文本（这样更快）
my $dumpflags = '-l system -c ~ -M';

open my $ELDUMP, '-|', "$eldump $dumpflags" or die "Unable to run $eldump:$!\n";

print 'Reading system log.';

my ( $date, $time, $source, $type, $category, $event, $user, $computer );
while ( defined ( $_ = <$ELDUMP> ) ) {
    ( $date, $time, $source, $type, $category, $event, $user, $computer ) =
        split('~');
    $type{$source}++;
    print '.';
}
print "done.\n";

close $ELDUMP;

# 针对每种类型的事件，打印出其来源和每个来源的事件数量
foreach $type (qw(Error Warning Information AuditSuccess AuditFailure))
{
    print '-' x 65, "\n";
    print uc($type) . "s by source:\n";
    for ( sort keys %$type ) {
        print "$_ (${$type{$_}})\n";
    }
}
print '-' x 65, "\n";

```

下面是部分输出：

```

ERRORs by source:
BROWSER (8)

```



```

print "-- scanning for first host contacts from $user --\n";
my %contacts = ();# 保存联系过指定用户的主机名
while ( my $entry = User::Utmp::getutxent() ) {
    if ( $entry->{ut_user} eq $user ) {
        next if ( defined $ignore and $entry->{ut_host} =~ /$ignore/o );
        if ( $entry->{ut_type} == USER_PROCESS
            and !exists $contacts{ $entry->{ut_host} } )
        {
            $contacts{ $entry->{ut_host} } = $entry->{ut_time};
            print form $format, $entry->{ut_user}, $entry->{ut_host},
                scalar localtime( $entry->{ut_time} );
        }
    }
}

print "-- scanning for other contacts from those hosts --\n";
User::Utmp::setutxent();    # 重置到数据库开头

while ( my $entry = User::Utmp::getutxent() ) {
    # 在查找该主机时，如果发现它是某个用户的进程，且该连接
    # 是来自用户而不是被入侵的账户，那么输出该记录
    if ( $entry->{ut_type} == USER_PROCESS
        and exists $contacts{ $entry->{ut_host} }
        and $entry->{ut_user} ne $user )
    {
        print form $format, $entry->{ut_user}, $entry->{ut_host},
            scalar localtime( $entry->{ut_time} );
    }
}
User::Utmp::endutxent();    # 关闭数据库（这一步不是必需的）

```

该程序首先会扫描整个 *wtmpx* 数据，查找所有被入侵用户的登录记录。找到这些记录后，它会构建一个哈希来保存所有发起登录的机器的主机名。接着，它重新回到数据库开头，这样下次就可以从文件的开头开始进行扫描。第二次扫描查找出和那个主机名列表中的所有主机相关的连接，一旦找到就打印出该记录。我们可以很容易地修改这个程序，让它能扫描保存在某个目录中的轮循的 *wtmp* 日志文件。要实现这样的功能，我们只需要遍历整个列表，依次对各个文件名调用 `User::Utmp::utmpxname()` 函数就可以了。

这个程序有个问题就是它的针对性太强。也就是说，它只会进行精确的主机名匹配。如果入侵者来自某个ISP给DSL或有线调制解调器分配动态地址的地址池（一般都是这样），则有可能每次它发起连接的主机名都不一样。尽管如此，这种不完美的解决方法在很多情况下还是很有帮助的。

除了简单性，我们所讨论的流式读取—统计方式还具有快速及内存占用小的优点。它能和本章早先提到的无状态型的日志文件很好地一起工作。但有时候，尤其是当要处理有状态的数据时，我们需要使用不同的解决方法。

读取－记录－处理

和之前我们提到的方式完全相反，现在我们要介绍的方式会尽快将数据传进来，将它们读进内存并在读取完成后进行处理。让我们看一些关于这个策略不同版本的实现。

首先介绍一个简单的例子：假设我们有一个FTP传输日志，我们想知道最频繁地被传输的文件是哪些。下面有一些来自wu-ftp服务器传输日志的记录样例。为了更容易看清日志行的开始和结尾，我们在记录之间加上了空行：

```
Sun Dec 27 05:18:57 2008 1 nic.funet.fi 11868
/net/ftp.funet.fi/CPAN/MIRRORING.FROM a _ o a cpan@perl.org ftp 0 *

Sun Dec 27 05:52:28 2008 25 kju.hc.congress.ccc.de 269273
/CPAN/doc/FAQs/FAQ/PerlFAQ.html a _ o a mozilla@ ftp 0 *

Sun Dec 27 06:15:04 2008 1 rising-sun.media.mit.edu 11868
/CPAN/MIRRORING.FROM b _ o a root@rising-sun.media.mit.edu ftp 0 *

Sun Dec 27 06:15:05 2008 1 rising-sun.media.mit.edu 35993
/CPAN/RECENT.html b _ o a root@rising-sun.media.mit.edu ftp 0 *
```

表10-4 列出了输出行中的所有字段（请参考wu-ftp服务器的手册页xferlog(5)来获取每个字段的详细信息）。

表10-4：wu-ftp服务器传输日志中的字段

字段号 字段名

0	current-time
1	transfer-time（单位：秒）
2	remote-host
3	filesize
4	filename
5	transfer-type
6	special-action-flag
7	direction
8	access-mode
9	username
10	service-name
11	authentication-method
12	authenticated-user-id

下面的代码用来显示哪些文件最频繁地被传输：

```

my $xferlog = '/var/adm/log/xferlog';
my %files = ();

open my $XFERLOG, '<', $xferlog or die "Unable to open $xferlog:$!\n";

while (defined ($line = <$XFERLOG>)){
    $files{(split(' ', $line))[8]}++;
}

close $XFERLOG;

for (sort {$files{$b} <=> $files{$a} || $a cmp $b} keys %files){
    print "$_: $files{$_}\n";
}

```

我们读取该文件的每一行，然后将文件名作为哈希的键，通过增加该键相应的键值来记录文件被传输的次数。我们用`split()`函数对每一行日志的内容进行分割并返回一个列表，然后通过数组下标将文件名提取出来^[注6]：

```
$files{(split)[8]}++;
```

你也许注意到了，我们指定的元素（8）和在表10-4中所列出的第8个字段不同。没办法，这是由于原始文件中缺乏字段分隔符导致的。由于我们是按空白字符（`split()`默认的分隔符）来进行分割的，所以上面文件中的日期时间部分就被分割成了5个独立的列表项。

在这段代码中有一个微妙的小技巧，那就是匿名的`sort`函数，我们用它来对值进行排序：

```
for (sort {$files{$b} <=> $files{$a} || $a cmp $b} keys %files){
```

注意在第一部分中，`$a`和`$b`是按字母顺序倒序排列的。它使`sort`以倒序的方式返回各项记录，这样就能最先显示出最频繁被传输的文件。匿名`sort`函数的第二部分（`|| $a cmp $b`）确保了那些被传输次数相同的文件会按文件名被排序。

如果我们想限制这个脚本只针对特定的文件或文件夹进行统计，我们可以让用户指定一个正则表达式作为脚本的第一个参数。举例来说，在`while`循环中加上如下的代码：

```
next unless /$ARGV[0]/o;
```

这就允许我们指定一个正则表达式来限制哪些日志行会被统计进来。

注6：如果有文件名包含空白字符，那么`split()`函数的返回结果可能会和我们所期望的不同。在这种情况下我们可能需要换用正则表达式来对记录行进行分割。

正则表达式

构造正则表达式通常是日志解析最重要的部分。正则表达式像程序中的筛子一样，帮我们从日志中提取出感兴趣的数据。本章中使用到的正则表达式是比较基础的，但你自己实际使用时可能会需要创建更复杂的正则表达式。为了更有效地利用它，我们可能需要使用第8章中提到的正则表达式技术。

Jeffrey Friedl的《Mastering Regular Expressions》（O'Reilly出版）一书是学习正则表达式最好的资源。花精力学习正则表达式能让我们在很多方面都获益。

让我们看看读取－记录－处理这种方式的另外一个例子，它使用了前一节中的“breach-finder”程序。我们前面的代码只显示出入侵者成功的登录记录。如果我们想找出失败的登录尝试呢？为了获得这些信息，我们需要引入另外一个日志文件。

警告： 这种情况也暴露了Unix的一个缺陷：Unix系统倾向于将日志信息以不同的格式存储在很多不同的地方。系统只提供了很少的工具来处理这种不一致性（幸运的是，我们有Perl）。为了解决之前提到的问题，我们常常需要多个数据源。

在本例中，对我们最有用的日志文件是由Wietse Venema的Unix安全工具*tcpwrappers*通过*syslog*生成的。*tcpwrappers*提供了守护程序和程序库来控制对网络服务的访问。像*telnet*这样独立的网络服务都可以被配置，好让*tcpwrappers*程序能管理所有的网络连接。当尝试建立连接的时候，*tcpwrappers*程序会先通过*syslog*记录它，然后再将该连接请求转给真正的服务器，或者采取其他的动作（如丢掉该连接）。关于是否允许连接通过，这个取决于一些简单的用户规则（比如说只允许从特定主机来的连接）。*tcpwrappers*还能通过DNS反向查找来初步地判定连接是否来自它所声称的地方。它甚至还能被配置为在可能的情况下对发起连接的用户名进行记录（通过RFC 931识别协议）。如果需要更多*tcpwrappers*的详细描述，请参阅Simson Garfinkel、Gene Spafford及Alan Schwartz合著的《Practical Unix & Internet Security》（O'Reilly出版）。

对我们而言，我们只需要给之前的breach-finder程序添加一些代码，让它扫描*tcpwrappers*的日志（本例中为*tcpdlog*文件）来检查和所有之前在扫描*wtmp*时搜集的可疑主机列表相关的连接。我们将下面的代码加到之前代码的尾部：

```
# tcpd 日志文件的路径
my $tcpdlog = '/var/log/tcpd/tcpdlog';

print "-- connections found in tcpdlog --\n";
open my $TCPDLOG, '<', $tcpdlog or die "Unable to read $tcpdlog:!\n";
my ( $connecto, $connectfrom );
while ( defined( $_ = <$TCPDLOG> ) ) {
```



```

next if !/connect from /;    # 我们只关心连接
( $connecto, $connectfrom ) = /(.):s+connect from\s+(.+)/;
$connectfrom =~ s/^.+@//;

print
    if ( exists $contacts{$connectfrom}
        and $connectfrom !~ /$ignore/o );
}
close $TCPDLOG;

```

我们会得到如下的输出：

```

-- first host contacts from baduser --
user      hostxx.ccs.example.edu Thu Apr  3 13:41:47 2008
-- other connects from suspect machines --
user2     hostxx.ccs.example.edu Thu Oct  9 17:06:49 2008
user2     hostxx.ccs.example.edu Thu Oct  9 17:44:31 2008
user2     hostxx.ccs.example.edu Fri Oct 10 22:00:41 2008
user2     hostxx.ccs.example.edu Wed Oct 15 07:32:50 2008
user2     hostxx.ccs.example.edu Wed Oct 22 16:24:12 2008
-- connections found in tcpdlog --
Jan 12 13:16:29 host2 in.rshd[866]: connect from user4@hostxx.ccs.example.edu
Jan 13 14:38:54 host3 in.rlogind[4761]: connect from user5@hostxx.ccs.example.edu
Jan 15 14:30:17 host4 in.ftpd[18799]: connect from user6@hostxx.ccs.example.edu
Jan 16 19:48:19 host5 in.ftpd[5131]: connect from user7@hostxx.ccs.example.edu

```

你也许会注意到，这个输出里面包含了两个不同时间范围的连接：之前在*wtmpx*中找到的连接是4月3号到10月22号的，而*tcpwrappers*的数据显示的只有1月份的连接。这种差异说明我们的*wtmpx*文件和*tcpwrappers*文件各自进行轮循的速度不同。相对于理所应当认为这两个日志文件在时间上是相互对应的，我们在写代码的时候需要十分小心这些细节问题。

最后一个也是最复杂的采用读取－记录－处理方式的例子，让我们来看一个需要结合有状态数据和无状态数据的任务。如果我们想要更全面地了解*wu-ftpd*服务器活动的状况，可能需要编写代码来结合机器的*wtmp*文件所记录的登录和注销活动，以及记录在*xferlog*文件中的文件传输信息。如果我们能通过查看输出来获知什么时候FTP会话开始和结束、在该会话期间有哪些传输发生，那就最好不过了。

下面的一段输出样本来自我们将要进行组合的代码。它显示了四个3月份的FTP会话。第一个会话显示有一个文件被传输到机器上；下面两个会话显示有文件从那台机器被传输到外面；最后一个会话显示有一条连接但无任何文件传输发生：

```

Thu Mar 12 18:14:30 2008-Wed Mar 12 18:14:38 2008 pitpc.host.ed
-> /home/dnb/makemod

Sat Mar 14 23:28:08 2008-Fri Mar 14 23:28:56 2008 traal-22.host.edu
<- /home/dnb/.emacs19

```

```
Sat Mar 14 23:14:05 2008-Fri Mar 14 23:34:28 2008 traal-22.host.edu
  <- /home/dnb/lib/emacs19/cperl-mode.el
  <- /home/dnb/lib/emacs19/filladapt.el

Wed Mar 25 21:21:15 2008-Tue Mar 25 21:36:15 2008 traal-22.host.edu
  (no transfers in xferlog)
```

要产生这样的输出是不容易的，因为我们需要将无状态的数据归入有状态的日志中。*xferlog*传输日志只显示时间和发起传输的主机名。*wtmpx*日志显示从其他主机到服务器的连接和断开连接的记录。下面我们看看如何用读取－记录－处理的方式将这两种类型的数据结合起来。首先，我们要为程序定义一些变量并加载一些辅助模块：

```
use Time::Local; # 用来做日期到Unix时间（从纪元开始到现在的秒数）的转换
use User::Utmp qw(:constants);
use Readonly;    # 为了增强可读性，创建只读常量

# 传输日志的路径
my $xferlog = '/var/log/xferlog';

# wtmpx 日志的路径
my $wtmpx = '/var/adm/wtmpx';

# 将每个月的名字映射到数字
my %month = qw{Jan 0 Feb 1 Mar 2 Apr 3 May 4 Jun 5 Jul 6
  Aug 7 Sep 8 Oct 9 Nov 10 Dec 11};
```

现在，让我们看看读取*wu-ftpd*的*xferlog*日志的过程：

```
# 扫描 wu-ftpd 传输日志并填充 %transfers 数据结构
print 'Scanning $xferlog...';
open my $XFERLOG, '<', $xferlog or die "Unable to open $xferlog:$!\n";

# 我们将要从日志中解析出的字段
my ( $time, $rhost, $fname, $direction );
my ( $sec, $min, $hours, $mday, $mon, $year );
my $unixdate; # the converted date
my %transfers; # our data structure for holding transfer info

while (<$XFERLOG>) {

  # 使用数组切片来选择我们需要的字段
  ( $mon, $mday, $time, $year, $rhost, $fname, $direction )
    = (split)[ 1, 2, 3, 4, 6, 8, 11 ];

  $fname =~ tr/ -~//cd; # remove "bad" chars
  $rhost =~ tr/ -~//cd; # remove "bad" chars

  # 'i' 代表“传入的”
  $fname = ( $direction eq 'i' ? '->' : '<- ' ) . $fname;

  # 将传输时间转换为 Unix epoch 时间格式
  ( $hours, $min, $sec ) = split( ':', $time );
  $unixdate = timelocal( $sec, $min, $hours, $mday, $month{$mon}, $year );
```

```

# 将数据存入包含列表的列表的哈希中，也就是：
# $transfers{hostname} = ( [time1, $filename1],
#                           [time2, $filename2],
#                           ...)
push( @{ $transfers{$rhost} }, [ $unixdate, $fname ] );
}
close $XFERLOG;
print "done.\n";

```

前面这段代码中有三行Perl代码需要解释。前两行是：

```

$fname =~ tr/ -~//cd;    # 删除“不好的”字符
$rhost =~ tr/ -~//cd;    # 删除“不好的”字符

```

这么做只是不让乱七八糟的东西在程序稍后的输出结果中出现。这行代码可以去掉文件名中所有的控制字符，所以当某个用户传输了一个文件名怪怪的文件时（不管是无意还是恶意为之），在打印这个文件名的时候就不会出现问题，我们的终端程序也不会出现奇怪的现象。在稍后的代码中，也会对读入的主机名做类似的清理工作。如果想全面地解决这个问题（应该这么做），我们可以写一段正则表达式，只接受“合法”的文件名，然后使用这段正则表达式所匹配的内容。但就目前而言，我们还是用上面的方法。

另外一行更复杂的Perl代码是push()语句：

```

push( @{ $transfers{$rhost} }, [ $unixdate, $fname ] );

```

这行创建了一个哈希，它包含了列表的列表，看起来像下面这样：

```

$transfers{hostname} =
    ([time1, filename1], [time2, filename2], [time3, filename3]...)

```

%transfers哈希以发起传输的主机名作为键。

对于每一个主机，我们以列表的形式保存一对与传输相关的信息，每一对记录都包含了文件是什么时候传输的以及该文件的文件名。我们选择使用“自纪元起所经过的秒数”作为存储时间的格式^[注7]，这么做是为了让接下来的比较容易。Time::Local模块的timelocal()函数可以帮助我们将时间按标准进行转换。由于我们是按时间顺序对传输日志文件进行扫描的，因此列表中的元素对也是按时间顺序排列的（这个属性在接下来会很有用）。

让我们继续wtmpx的扫描：

```

# 扫描wtmpx文件并用扫描到的ftp会话填充@sessions结构
my ( %connections, @sessions );

```

注7： 这是自任意起点开始的秒数。比如说，Unix机器上的epoch就是1970年1月1号0时0分0秒的格林威治时间。

```

print "Scanning $wtmptx...\n";
User::Utmp::utmpxname($wtmptx);
while ( my $entry = User::Utmp::getutxent() ) {
    next if ( $entry->{ut_id} ne 'ftp' ); # 忽略非 ftp 会话

    # 用包含列表的列表的哈希来保存“打开”的连接（即 LoL, List of List, 在这里像
    # 栈一样被存在哈希里，以设备名作为键）
    if ( $entry->{ut_user} and $entry->{ut_type} == USER_PROCESS ) {
        $entry->{ut_host} =~ tr/ -~//cd; # remove "bad" chars

        push(
            @{ $connections{ $entry->{ut_line} } },
            [ $entry->{ut_host}, $entry->{ut_time} ]
        );
    }

    # 发现关闭连接的记录，尝试和打开连接的记录配对
    if ( $entry->{ut_type} == DEAD_PROCESS ) {
        if ( !exists $connections{ $entry->{ut_line} } ) {
            warn "found lone logout on $entry->{ut_line}:"
                . scalar localtime( $entry->{ut_time} ) . "\n";
            next;
        }

        # 创建会话列表，每个会话都以这样的形式表示：
        # （主机名，登录时间，注销时间）
        push(
            @sessions,
            [
                @ shift @{ $connections{ $entry->{ut_line} } },
                $entry->{ut_time}
            ]
        );

        # 如果在该终端没有更多其他连接了，就将它从哈希中删除
        delete $connections{ $entry->{ut_line} }
            unless ( @ { $connections{ $entry->{ut_line} } } );
    }
}
User::Utmp::endutxent();
print "done.\n";

```

让我们看看这段代码做了什么。它一次一条地从`wtmptx`中读取记录。如果当前的记录是在特定的设备名（即`ftp`）上发生的，我们就知道这是个FTP会话。指定一条`wtmptx`数据库中的`ftp`记录，我们可以查看它描述的是FTP会话中连接的打开（即`ut_type`为`USER_PROCESS`）还是连接的关闭（即`ut_type`为`DEAD_PROCESS`）。

如果它描述的是连接的打开，我们就将该信息存储到一个数据结构中，这个数据结构专门保存打开状态的会话，即`%connections`哈希。和前面子例程中的`%transfers`哈希一样，它也是个包含了列表的列表的哈希，但这次它是以各个连接的设备（即`tty/pty`）作为键。这个哈希中的值包含了发起连接的主机名和建立时间这些详细信息。

为什么用如此复杂的数据结构来跟踪打开连接呢？不幸的是，*wtmpx*中没有简单的“打开－关闭 打开－关闭 打开－关闭”这样成对出现的记录行。比如说，看看下面这些*wtmpx*中的记录行（用本章早些时候的第一个处理*wtmpx*的程序打印出来的）：

```
ftpd1833:dnb:ganges.example.edu:Thu Mar 27 14:04:47 2008
ttyp7:dnb:(exit):Thu Mar 27 14:05:11 2008
ftpd1833:dnb:hotdiggitydog.example.edu:Thu Mar 27 14:05:20 2008
ftpd1833:dnb:(exit):Thu Mar 27 14:06:20 2008
ftpd1833:dnb:(exit):Thu Mar 27 14:06:43 2008
```

注意同一个设备上的两条打开FTP连接的记录（第1行和第3行）。如果仅仅只按设备来将单个连接保存在普通的哈希中，当我们找到第二条记录的时候，会丢失掉第一条记录。

相反地，我们将每个设备作为%connections哈希的键，将列表的列表作为值像一个栈一样保存在其中。当我们发现一条连接建立的时候，就将那个设备上（*host,login-time*）这样一对连接信息保存到该设备的栈中。每当我们发现同一个设备的关闭连接记录行时，我们就将相应的打开连接的记录从栈中“弹”出，并将该会话的完整信息存入另外一个数据结构，即@sessions数组中。这就是下面这条语句的目的：

```
push(
    @sessions,
    [
        @ { shift @ { $connections { $entry->{ut_line} } } },
        $entry->{ut_time}
    ]
);
```

让我们把这条语句从里边剥开，让它变得更清晰一些。加粗的部分会返回一个指向栈/列表的引用，它包含了在特定设备（*ut_line*）上打开连接的成对记录：

```
push(
    @sessions,
    [
        @ { shift @ { $connections { $entry->{ut_line} } } },
        $entry->{ut_time}
    ]
);
```

这里将对头一个连接记录的引用从栈中弹出：

```
push(
    @sessions,
    [
        @ { shift @ { $connections { $entry->{ut_line} } } },
        $entry->{ut_time}
    ]
);
```

我们将它解引用来获得实际的包含（*host,login-time*）这样一对信息的连接记录列表。如果我们把这对信息放在另外一个以连接时间结尾的列表中，Perl会用这对信息进行

行插值，然后我们得到了一个包含三个元素的列表。这样我们就有了包含三个信息即主机名、登录时间、注销时间的记录：

```
push(
    @sessions,
    [ @{ shift @{$connections{ $entry->{ut_line} } } },
      $entry->{ut_time}
    ]
);
```

现在，我们已经拥有了一个完整的FTP会话记录（即发起连接的主机、连接开始时间、断开时间）保存在单一列表中，所以我们可以将它放到一个匿名数组的引用中，然后保存到作为列表的列表的数组@sessions中，以备后用：

```
push(
    @sessions,
    [ @{$shift @{$connections{ $entry->{ut_line} } } },
      $entry->{ut_time}
    ]
);
```

感谢这条在程序中会被频繁使用的语句，现在我们有包含所需会话的列表。

要完成整个工作，我们需要检查对于某个设备，栈是不是为空（也就是说，没有更多连接打开请求）。如果是的，我们就可以将该设备在哈希中的相应记录删掉，因为我们知道连接已经结束了：

```
delete $connections{ $entry->{ut_line} }
unless ( @{$connections{ $entry->{ut_line} } } );
```

现在，该将我们的两个数据集结合起来了。针对每一个会话，我们想打印出上面所说包含三个信息的连接记录，以及在该会话中所传输的文件：

```
# 定义常量，让包含三个信息的连接记录的数据结构更可读；
# 列表中的元素位置像这样 ($HOSTNAME,$LOGIN,$LOGOUT)
Readonly my $HOSTNAME => 0;
Readonly my $LOGIN     => 1;
Readonly my $LOGOUT    => 2;

# 遍历整个会话日志，给会话和传输记录配对

foreach my $session (@sessions) {

    # 打印会话时间
    print scalar localtime( $session->[$LOGIN] ) . '- '
          . scalar localtime( $session->[$LOGOUT] ) . ' '
          . $session->[$HOSTNAME] . "\n ";

    # 返回指定登录会话中所有被传输的文件
```

```

# 处理本次登录中无传输发生的简单情况
if ( !exists $transfers{ $session->[HOSTNAME] } ) {
    print " \t( no transfers in xferlog ) \n ";
    next;
}

# 处理头一个传输记录不是发生在本次登录中的简单情况
if ( $transfers{ $session->[HOSTNAME] }->[0]->[0] > $session->[LOGOUT] )
{
    print " \t( no transfers in xferlog ) \n ";
    next;
}

my (@found) = ();    # 保存我们在每个会话中找到的传输记录

# 找到本次会话中传输的所有文件
foreach my $transfer ( @{ $transfers{ $session->[HOSTNAME] } } ) {

    # 若传输发生在登录前
    next if ( $transfer->[0] < $session->[LOGIN] );

    # 若传输发生在注销后
    next if ( $transfer->[0] > $session->[LOGOUT] );

    # 若我们已经报告过该条记录
    next if ( !defined $transfer->[1] );

    # 记录该传输通过对文件名 undef 来将其标记为已用
    push( @found, " \t " . $transfer->[1] . " \n " );
    undef $transfer->[1];
}
print( scalar @found ? @found : " \t( no transfers in xferlog ) \n" )
    . " \n ";
}

```

上面的代码先从排除简单的情况开始：如果我们没有发现该发起连接的主机有任何传输发生，或者和这个主机相关的头一个传输记录发生在本次会话之后，我们就可以判断在本次会话过程中无文件被传输。

如果我们不能排除简单情况，则需要查找整个传输记录列表。对于每一个该主机发起的传输，我们都会看看它是不是在本次会话开始和结束之间完成的，而该主机不满足这个条件的其他传输我们就不再测试了。还记得我提到过所有传输记录在数据结构中是以时间顺序排列的吗？这里我们就用到了这个属性。

最后一个用来测试某个传输记录是否合法的语句看起来有些特别：

```

# 看看我们是否已经用过该记录
next if ( !defined $transfer->[1] );

```

如果两个来自同一主机的匿名FTP会话在时间上重叠了，那么在这个时间片中我们就无法知道到底是哪个会话开始的传输，进行的文件上传或下载。在我们所有的日志中都没

有相关的信息能帮助我们进行这样的判断。我们所拥有的最好选择就是制定一个标准并遵守它。我们的代码中所遵循的标准就是“尽可能将传输记录归于最先配对的会话”，代码中前面的测试语句和紧接着的对文件做`undef`操作来标记的方法就确保了我们是遵守这个标准的。

如果最后一条测试通过了，我们就将该文件成功地加入到当前会话的文件传输列表中（即数组`@found`中），此时该会话和它对应的文件传输记录就被打印出来了。

读取—记录—处理型的程序要做这样的结合性操作会将事情变得很复杂，尤其是在当把数据源进行结合时过程是模糊不清的情况下。因此，为了遵循Perl的精神，让我们看看是否能找到一种简单易行的方式。

黑盒

在Perl的世界中，当你尝试编写一些常用的代码时，经常会发现其实已经有别人写了能完成同样任务的东西并分享出来。这种情况下，我们往往只要把所需的数据按事先约定好的方式传递给别人写好的模块，然后等着接收返回的结果就好了，至于模块是怎么完成任务的我们并不需要去关心。这种情况被称为“黑盒方式”。当然这种方式有它的风险，所以接下来请留心补充内容中的警告信息。

恭喜你！你成了模块的新维护者！

尽管我推荐多用黑盒方式，但使用它还是有风险的。让我给你敲敲警钟。

在本书的第一版时，我极力推荐了一个叫SyslogScan的模块。它是一个用来解析syslog很棒的模块，尤其是它对邮件传输代理程序sendmail产生的日志的支持非常好。它把所有麻烦活都搞定了，如解析原始sendmail日志，将日志记录行两两配对并和处理邮件的记录结合起来。它提供了让人喜爱的简洁的接口来一条条信息地遍历整个日志文件。遍历日志时的这些迭代对象可以交由该包的其他部分进行处理，可以生成摘要报告、摘要对象之类的东西。这些对象也能依次被传给该包的另外一些部分，可以生成更具有表现力的报告。它的工作做得十分优美。

但不知道从什么时候开始，sendmail的开发者对他们的日志文件的格式做了些许小修改。于是SyslogScan就没法像以前那样顺利地解析日志文件了，因此也就完全没法正常工作了。

在大多数情况下，这种改变不会引起太大麻烦，因为模块的作者会注意到这样的问题，然后针对日志文件格式的改变发布一个新的版本。但不幸的是，SyslogScan的作者貌似从1997年起就从Perl的世界里消失了。于是该模块到本书写作之时在CPAN上仍旧没有任何更新，成了一个不能用的模块。

如果日志格式有改变而我们仍依赖于该模块，那么我们有三种选择：

1. 开始用另外的模块（如果之前依赖的模块是唯一的模块，这个选择就不成立了）。
2. 编写我们自己的替代模块（可能需要做很多工作）。
3. 尝试针对日志格式的改变，自己给SyslogScan打补丁。

在这三种选择中，可能第三种选择需要做的工作最少。有可能只需要做少量的修改该模块就能重新工作。但从此以后，恭喜你，你现在成了这个模块的维护者了（至少在你自己的小环境中）！如果由于某种原因它又没法工作了，你就有责任再次对它进行修复。也许这对你算不上什么大问题，但在决定依赖他人代码前，了解一下可能存在的缺点也是值得的。

Perl社区的优势就是乐于分享代码。在CPAN上有很多用来解析日志的模块可以使用。这些模块中的大部分都是专门针对某种任务的。举例来说，Philippe “Book” Bruhat的Log::Procmail模块就是专门用来帮助我们更方便地遍历和解析procmail邮件过滤程序所产生的日志。要将我们收到的邮件来源地址列表和邮件文件保存在什么地方打印出来，我们只需要编写如下代码：

```
use Log::Procmail;

my $procl = new Log::Procmail '/var/log/procmail';

while (my $entry = $procl->next){
    print $entry->from . ' => ' . $entry->folder . "\n";
}
```

还有很多Apache日志文件的解析器（如Apache::ParseLog模块、Parse::AccessLogEntry模块和Apache::LogRegex模块）可以帮助我们解析Apache日志格式的文件。

还有一些模块可以用来构建满足我们自己特定需求的解析器。其中有些模块的“黑盒性”比其他模块要强。在Unix中，Parse::Syslog作为黑盒性质的模块，一直是用来解析syslog类的记录行比较好的选择。它还有另外一个非常棒的特性，Parse::Syslog模块的new()方法可以接受File::Tail对象，而不仅仅是司空见惯的文件句柄。通过该对象，Parse::Syslog便可以在文件仍被写入的同时对它进行操作，就像这样：

```
use File::Tail;
use Parse::Syslog;

my $file = File::Tail->new( name => '/var/log/mail/mail.log' );

my $syslg = Parse::Syslog->new( $file );
```

```

while ( my $parsed = $syslg->next ) {
    print $parsed->{host} . ':'
      . $parsed->{program} . ':'
      . $parsed->{text} . "\n";
}

```

如果你想用更基础的部件来构造自己的解析器，你可能需要查看一下帮助构造正则表达式的一系列模块。举例来说，Dmitry Karasik的`Regexp::Log::DateRange`模块可以帮助我们构造比较复杂的正则表达式来选择`syslog`文件中的日期范围：

```

use Regexp::Log::DateRange;

# 构建一个正则表达式以匹配 May 31 8:00a 到 May 31 11:00a
my $regexp = Regexp::Log::DateRange->new('syslog', [ qw(5 31 8 00) ],
                                           [ qw(5 31 11 00) ] );
# $regexp 现在包含: 'may\s+31\s+(?:0?[8-9]|10)\:\:|11\:0?0\:'
# 对该正则表达式进行编译以获得更好的性能
$regexp = qr/$regexp/i;

# 现在使用该正则表达式
if ($input =~ /$regexp/) { print "$input matched\n" };

```

如果想更进一步，Philippe “Book” Bruhat的`Regexp::Log`模块允许我们基于它去构造能帮我们创建正则表达式的模块。为了了解其派生的模块是如何工作的，最简单的方法就是看看基于它构建的一个派生模块。Barbie的`Regexp::Log::Common`模块就是一个很好的派生模块的例子，它被用来解析Common Log Format（通用日志格式，Apache等软件包用到的格式）的日志。

下面是像`Regexp::Log::Common`这样的派生模块的用法：

```

use Regexp::Log::Common;

my $rlc = Regexp::Log::Common->new( format => ':extended' );
$rlc->capture( qw(:none referer) );
my $regexp = $rlc->regexp;

# 现在我们拥有能捕获匹配针对 Extended Common Log Format 的每一条
# 记录行的 referer 字段的正则表达式，
# 即：
#      ($referer) = $logline =~ /$regexp/

```

加载该模块后，告诉它我们将处理包含Extended Common Log Format的记录行的文件（`:extended`是用来指定该格式中所有字段名的快捷方法，如果需要，我们也可以手动列出想要的字段名）。

接着，我们使用`capture()`来告诉模块哪些字段是我们想要捕获的。`capture()`可能看起来像是一个简单的方法调用，用以设置需要被捕获的字段列表，但实际上它是把这些字段添加到当前需要捕获的列表中去。这个列表开始的时候默认是包含所有的字段，所

以，在告诉它我们只需要捕获一个字段（“referer”）前，我们需要使用:none这个特殊的关键字来清空当前的字段列表。

为了以黑盒编程方法来结束本节，我们将介绍其中一个黑盒分析模块，有了该模块的帮助，编写日志分析模块就会简单很多。Alex White编写了一个叫做Log::Statistics的模块，可以实现简单的（基于统计方法的）日志文件分析。让我们看看它是如何工作的。

在加载完模块并创建好新对象后，第一步是告诉该模块如何把你的日志文件解析成字段。在本例中，我们将使用PureFtpd服务器（<http://www.pureftpd.com>）产生的stats日志文件格式。它具有如下的字段：

```
<date> <session id> <user> <ip> <U or D> <size> <duration> <file>
```

下面是三行样例（加入了额外的空行以便阅读），这样我们就对它是什么样子有所了解：

```
1151826125 44a778cc.1a41 ftp bb.67.1333.static.  
theplanet.com D 29 0 /home/ftp/net/mirrors/ftp.funet.fi/pub/  
languages/perl/CPAN/authors/02STAMP  
  
1151826483 44a77a32.1cf4 ftp ajax-1.apache.org D 11 0  
/home/ftp/net/mirrors/dev.apache.org/dist/DATE  
  
1151829011 44a78408.1eca ftp 69.51.111.252 D 1809 0 /home/ftp/net  
/mirrors/squid.nlanr.net/pub/squid-2/md5s.txt
```

为了解析这种类型的记录行，我们告诉Log::Statistics模块去使用一个自定义的正则表达式，这个正则表达式可以捕获记录的各个字段：

```
use Log::Statistics;  
  
my $ls = Log::Statistics->new();  
$ls->add_line_regexp(  
    '^(\\d+)\\s+(.*)\\s+(\\w+)\\s+(.*)\\s+(U|D)\\s+(\\d+)\\s+(\\d+)\\s+(.*)';
```

接下来，我们告诉模块哪些字段是摘要中要包含的，以及这些字段在正则表达式的什么位置被捕获：

```
$ls->add_field( 3, 'ip' );  
$ls->add_field( 4, 'direction' );
```

剩下的代码完成日志读取和解析的工作：

```
open my $LOG, '<', 'pureftpd.log';  
  
my $line = '';  
while ( defined ($line = <$LOG> ) ) {  
    $ls->parse_line($line);
```

```

}

close($LOG);
print $ls->get_xml();

```

最后的结果是一个基于XML的报告，看起来如下：

```

<?xml version="1.0" standalone="yes"?>

<log-statistics>
  <fields name="direction">
    <direction name="D" count="4674" />
  </fields>
  <fields name="ip">
    <ip name="0x530f53.hrnxx2.adsl-dhcp.tele.dk" count="2" />
    <ip name="12.135.144.8" count="6" />
    <ip name="12.24.221.254" count="1" />
    <ip name="124.14.4.223" count="1" />
    <ip name="125.13.133.183" count="2" />
    ...
  </fields>
</log-statistics>

```

在这份报告中，我们可以看到有4 674次下载被记录；我们还获得了一张列表，其中包含进行这些下载的IP地址或主机名以及各自下载的次数。如果希望报告更漂亮些，包含各个主机所下载的文件，我们可以把add_field()做如下修改：

```

$ls->register_field( 'ip', 3 );
$ls->register_field( 'file', 7 );
$ls->add_group(['ip','file']);

```

前两行将文件名和正则表达式中匹配的相应位置关联起来（和add_field()相似，并不对它们进行统计），最后一行指定在计算统计结果的时候按什么字段进行分组：

```

<?xml version="1.0" standalone="yes"?>

<log-statistics>
  <groups name="ip-file">
    <ip name="12.135.144.8">
      <file name="/home/ftp/net/mirrors/ftp.funet.fi/pub/languages/perl/CPAN/
authors/01mailrc.txt.gz" count="1" />
      <file name="/home/ftp/net/mirrors/ftp.funet.fi/pub/languages/perl/CPAN/
authors/id/H/HA/HAKANARDO/CHECKSUMS" count="1" />
      <file name="/home/ftp/net/mirrors/ftp.funet.fi/pub/languages/perl/CPAN/
authors/id/Y/YV/YVES/CHECKSUMS" count="1" />
      <file name="/home/ftp/net/mirrors/ftp.funet.fi/pub/languages/perl/CPAN/
authors/id/Y/YV/YVES/MIME-Lite-3.01.tar.gz" count="1" />
      <file name="/home/ftp/net/mirrors/ftp.funet.fi/pub/languages/perl/CPAN/
modules/02packages.details.txt.gz" count="1" />
      <file name="/home/ftp/net/mirrors/ftp.funet.fi/pub/languages/perl/CPAN/
modules/03modlist.data.gz" count="1" />
    </ip>
  </groups>
</log-statistics>

```

```
...
</groups>
</log-statistics>
```

想知道是谁下载了哪些文件？我们只需要简单地将前面代码中最后一行反转一下：

```
$ls->add_group(['file', 'ip']);
```

现在输出的内容就包含了下面这段：

```
...
<groups name="file-ip">
  <file name="/home/ftp/ls-lR.gz">
    <ip name="dau.cgr.ru" count="1" />
  </file>
  <file name="/home/ftp/net/mirrors/dev.apache.org/dist/DATE">
    <ip name="ajax-1.apache.org" count="14" />
    <ip name="month.cs.uu.nl" count="20" />
    <ip name="minotaur.apache.org" count="26" />
  </file>
  ...
</groups>
```

这个XML的输出可以被转换成漂亮的表格（比如用XSLT样式表），或者经过解析描绘成漂亮的图片。

如果想用现有模块来帮我们完成这类统计工作，有些模块还是值得看一看的（包括Algorithm::Accounting和Logfile）。若要在你的下一个项目中使用这些模块，务必请先了解一下该模块。

作为本节结尾，我需要提醒你要小心谨慎地对待黑盒方式。使用该方式的好处是我们常常能以更少的代码量来完成工作，这要感谢模块或脚本作者的辛勤劳动。而不好的方面是我们需要信任其他作者的代码。它有可能导致难以琢磨的bug或者不利于按我们的需求进行扩展。因此在应用于生产环境前最好先全面地审查它的代码。

使用数据库

我们最后要介绍的方式所需要的知识大部分属于Perl领域之外。因此，我们只会进行简单的技术方面的介绍，当然随着时间的推移所介绍的技术可能会变得更具普遍性。

前面的例子在具有适当内存的机器上能处理相当大量的数据集，但它不具备扩展性。假如我们有巨大的数据量，尤其是如果数据来自不同数据源，数据库自然是首选的工具。

在Perl中至少有两种方式来使用数据库。第一种是我称之为“Perl-only”的方法。使用这种方法时，所有的数据库活动都发生在Perl中，或者其程序库和Perl是紧耦合的。第二种方式使用类似于DBI家族的Perl模块，将Perl作为其他数据库的客户端，如

MySQL、Oracle或Microsoft SQL Server。让我们看看用这两种方式进行日志处理和分析的例子。

使用Perl-only的数据库。只要数据集不是特别大，我们可能只需要使用Perl-only的解决方案就足够了。我们将对之前的breach-finder程序进行扩展来作为例子。如果想找出我们的任何机器上入侵者的登录信息，该如何做？

第一步是将我们的机器上所有wtm_{px}数据导入至某种数据库中。就该例子而言，我们假设所有有问题的机器都能通过某种网络文件系统直接访问某个共享文件夹，像NFS一样。在处理之前，我们还需要选择一种数据库格式。

我所选择的“Perl数据库格式”是Berkeley DB格式。之所以“Perl数据库格式”用引号，是因为尽管对该数据库的支持是Perl源码自带的，但实际的数据库库文件需要从另外的来源获得（<http://www.oracle.com/database/berkeley-db/index.html>），并且需要在Perl编译对该数据库的支持功能之前安装好它。表10-5对Perl支持的不同数据库格式进行了比较。

表10-5：Perl支持的数据库格式之间的比较

	Unix	Windows	Mac OS X	键或值的 大小限制	字节序 是否无关
名称	支持	支持	支持		
“old” dbm	Yes	No	No	1K	No
“new” dbm	Yes	No	Yes	4K	No
Sdbm	Yes	Yes	Yes	1K（默认）	No
Gdbm	Yes ^a	Yes ^b	Yes ^a	None	No
DB	Yes ^a	Yes ^b	Yes ^a	None	Yes

a. 实际的数据库库文件可能需要单独下载。

b. 数据库库文件和Perl模块必须从Web下载（<http://www.roth.net>有个老版本，否则你需要使用Perl的Cygwin发行版）。某种程度上，或许你也能使用Strawberry Perl（参见第1章）。

我喜欢Berkeley DB格式，因为它能处理大量数据并且它是字节序无关的。对于我们将要看到的Perl代码来说，字节序无关是非常重要的特性，因为我们需要从不同的机器对同一个文件进行读写，而这些机器的架构可能是不同的。如果字节序对你来说非常重要，而你又不想编译并链接外部的程序库，DBM::Deep可以作为另一个不错的选择。

作为开始，我们首先需要填充数据库。为了简单性和可移植性，我们调用*last*程序来避免对多个不同的wtm_{px}文件执行unpack()。下面是代码，随后会有说明：

```

use DB_File;
use FreezeThaw qw(freeze thaw);
use Sys::Hostname;
use Fcntl;
use strict;

# 对于 Solaris, 如果不想主机名被截断, 可以使用 last -a, 但这就需要
# 修改下面的字段解析代码
my $lastex = '/bin/last' if ( -x '/bin/last' );
my $lastex = '/usr/ucb/last' if ( -x '/usr/ucb/last' );

my $userdb = 'userdata';
my $connectdb = 'connectdata';
my $thishost = &hostname;

open my $LAST, '-|', "$lastex" or die "Can't run the program $lastex:$!\n";

my ( $user, $tty, $host, $day, $mon, $date, $time, $when );
my ( %users, %connects );
while ( defined( $_ = <$LAST> ) ) {
    next if /^reboot/ or /^shutdown/ or /^ftp/ or /^account/ or /^wtmp/;
    ( $user, $tty, $host, $day, $mon, $date, $time ) = split;
    next if $tty =~ /^:0/ or $tty =~ /^console$/;
    next if ( length($host) < 4 );
    $when = $mon . ' ' . $date . ' ' . $time;

    push( @{$users{$user}}, [ $thishost, $host, $when ] );
    push( @{$connects{$host}}, [ $thishost, $user, $when ] );
}

close $LAST;

tie my %userdb, 'DB_File', $userdb, O_CREAT | O_RDWR, 0600, $DB_BTREE
or die "Unable to open $userdb database for r/w:$!\n";

my $userinfo;
for my $user ( keys %users ) {
    if ( exists $userdb{$user} ) {
        ($userinfo) = thaw( $userdb{$user} );
        push( @{$userinfo}, @{$users{$user}} );
        $userdb{$user} = freeze $userinfo;
    }
    else {
        $userdb{$user} = freeze $users{$user};
    }
}

untie %userdb;

tie my %connectdb, 'DB_File', $connectdb, O_CREAT | O_RDWR, 0600, $DB_BTREE
or die "Unable to open $connectdb database for r/w:$!\n";

my $connectinfo;
for my $connect ( keys %connects ) {
    if ( exists $connectdb{$connect} ) {

```

```

        ($connectinfo) = thaw( $connectdb{$connect} );
        push( @{$connectinfo}, @{$connects{$connect}} );
        $connectdb{$connect} = freeze($connectinfo);
    }
    else {
        $connectdb{$connect} = freeze( $connects{$connect} );
    }
}
untie %connectdb;

```

我们的代码获取`last`程序的输出并做了如下工作：

1. 过滤掉无用的记录行。
2. 将输出存到两个哈希中，这两个哈希包含的数据结构为列表的列表，看起来如下：

```

$users{username} =
    [[current host, connecting host, connect time],
     [current host, connecting host, connect time]
    ...
    ];
$connects{host} =
    [[current host, username1, connect time],
     [current host, username2, connect time],
     ...
    ];

```

将该数据结构放在内存中，然后试图将它合并进数据库中。

最后一步是最有趣的地方，所以让我们仔细看看。我们将哈希`%userdb`和`%connectdb`与数据库文件进行绑定^[注8]。这个魔法让我们能透明地访问这些哈希，而Perl则处理背后的从数据库文件中存储和获取数据的工作。但哈希只存储简单的字符串，我们如何将“包含列表的列表的哈希”这样一个数据结构存到一个哈希值中呢？

Ilya Zakharevich的`FreezeThaw`模块可以用来将我们的复杂数据结构存储到单个标量中，该标量可以被用作单个哈希的值。`FreezeThaw`可以接受任意Perl数据结构并将它编码成一个字符串。还有其他的模块也具有类似的功能，包括Gurusamy Sarathy的`Data::Dumper`模块（Perl自带）和Raphael Manfredi的`Storable`模块，但是`FreezeThaw`模块对复杂数据结构的表示最为紧凑（所以这里我们使用它）。这些模块都有各自的优势，所以如果你有类似这样的任务需要完成，请研究一下这三个模块。

在我们的程序中，我们会检查针对某个用户或主机的记录是否存在。如果它不存在，我们就简单地将该数据结构“冻结”（freeze）至一个字符串，并通过绑定的哈希来将该

注8：使用`DB_File`时不必总是使用`BTree`的存储格式，但该程序可以存储一些很长的值。这些值会让1.85版本的`DB_HASH`存储方法在测试时抛出异常（导致数据损坏），而`BTree`存储方法可以处理这种情况。1.85以后版本的DB库可能就不存在这样的bug了。

字符串存储到数据库中。如果它存在，我们就将找到的已存在的数据结构从数据库中“解冻”（thaw）回内存，附加上我们的数据，然后重新冻结并重新存储。

如果我们在多台机器上运行该代码，将会得到包含潜在有用信息的数据库，它对我们开发下一版的breach-finder程序能提供帮助。

注意：在 *wtmp* 文件的日志轮循完成之后是进行数据库填充的最合适的时机。

此处数据库填充的代码对于生产用途来说显得不靠谱。一个显著的缺陷是它不具备能力来防范多个程序实例同时对该数据库进行更新。而在NFS文件系统中对该文件使用文件锁是很冒险的，还不如从另外一个上层程序中调用该段代码，让那个上层程序来将各个机器上的信息收集过程序列化。

现在我们有填满数据的数据库，让我们看看使用这些信息改进后的breach-finder程序是什么样的：

```
use DB_File;
use FreezeThaw qw(freeze thaw);
use Perl6::Form;
use Fcntl;

my ( $user, $ignore ) = @ARGV;

my $userdb      = 'userdata';
my $connectdb   = 'connectdata';
my $hostformat  = '{<<<<<<<<<<<<} -> {<<<<<<<<<<<<} on {<<<<<<<<<<}';
my $userformat  = '{<<<<<<}: {<<<<<<<<<<<<} -> {<<<<<<<<<<<<} on {<<<<<<<<<<}';

tie my %userdb, 'DB_File', $userdb, O_RDONLY, 666, $DB_BTREE
or die "Unable to open $userdb database for reading:$!\n";

tie my %connectdb, 'DB_File', $connectdb, O_RDONLY, 666, $DB_BTREE
or die "Unable to open $connectdb database for reading:$!\n";
```

上面的代码中，我们加载好所需的模块，获取输入，设置好几个变量，并将它们绑定至我们的数据库文件。接下来需要做一些工作：

```
# 如果我们从未看见来自该用户的连接, 则可以退出
if ( !exists $userdb{$user} ) {
    print "No logins from that user\n";
    untie %userdb;
    untie %connectdb;
    exit;
}

my ($userinfo) = thaw( $userdb{$user} );

print "-- first host contacts from $user --\n";
my %otherhosts;
```

```

foreach my $contact ( @{$userinfo} ) {
    next if ( $ignore and $contact->[1] =~ /$ignore/o );
    print form $hostformat, $contact->[1], $contact->[0], $contact->[2];
    $otherhosts{ $contact->[1] } = 1;
}

```

这段代码说：如果我们已经看到过该用户，我们就使用thaw()对内存中该用户的联系记录进行重组。对于每一次联系，我们都会测试看看我们是否被要求忽略主机来源。如果不是，我们就将本次联系打印出来，并将来源主机记录到%otherhosts哈希中。

作为一种简单的方式，我们在这里使用哈希来从联系记录中收集包含唯一主机名的列表。现在我们已经拥有了包含入侵者可能连接过的主机列表，我们需要查出还有其他哪些用户也连接过这些潜在的被入侵了的主机。

要找出这些信息很容易，因为我们已经记录下来哪些用户登录过哪些机器，相反过程（也就是哪些机器被哪些用户登录过）的信息我们也记录在另外一个数据库文件中。现在我们可以查看所有在前一步中识别出的主机的相关记录。如果我们没有被要求忽略某个主机，且存在它的连接记录，则将登录过该主机的非重复用户名的列表记录至%userseen哈希中。

```

print "-- other connects from suspect machines --\n";
my %userseen;
foreach my $host ( keys %otherhosts ) {
    next if ( $ignore and $host =~ /$ignore/o );
    next if ( !exists $connectdb{$host} );

    my ($connectinfo) = thaw( $connectdb{$host} );

    foreach my $connect ( @{$connectinfo} ) {
        next if ( $ignore and $connect->[0] =~ /$ignore/o );
        $userseen{ $connect->[1] } = 1;
    }
}

```

在这三步处理过程中，最后一步完成得很漂亮。我们重新返回到之前的用户数据库来找出所有来自可疑机器的可疑用户所发起的连接：

```

foreach my $user ( sort keys %userseen ) {
    next if ( !exists $userdb{$user} );

    ($userinfo) = thaw( $userdb{$user} );

    foreach my $contact ( @{$userinfo} ) {
        next if ( $ignore and $contact->[1] =~ /$ignore/o );
        print form $userformat, $user, $contact->[1], $contact->[0],
            $contact->[2]
            if ( exists $otherhosts{ $contact->[1] } );
    }
}

```

剩下的就只剩清理工作啦：

```
untie %userdb;
untie %connectdb;
```

下面是该程序的输出样本（为保护隐私，用户名和主机名已被修改）：

```
-- first host contacts from baduser --
badhost1.example -> machine1.hogwarts.ed on Jan 18 09:55
ba
dhost2.example -> machine2.hogwarts.ed    on Jan 19 11:53
-- other connects from suspect machines --
baduser2: badhost1.example -> machine2.hogwarts.e on Dec 15 13:26
baduser2: badhost2.example -> machine2.hogwarts.e on Dec 11 12:45
baduser3: badhost1.example -> machine1.hogwarts.e on Jul 13 16:20
baduser4: badhost1.example -> machine1.hogwarts.e on Jun 9 11:53
baduser:  badhost1.example -> machine1.hogwarts.e on Jan 18 09:55
baduser:  badhost2.example -> machine2.hogwarts.e on Jan 19 11:53
```

这是个漂亮的程序范例，但它实际上没法扩展至大一些的机器集群。每次运行该程序，它都可能需要从数据库中读取一条记录，将其`thaw()`回内存中，添加新数据至该机器，再次`freeze()`它并存储回数据库中。这些操作是CPU时间和内存密集型的。整个过程可能在每次用户和机器连接发生的时候都会发生，所以它很快会变得很慢。

使用Perl作为客户端的SQL数据库。如果我们的数据集十分庞大，可能需要将我们的数据导入至更高级的SQL数据库中（商业的或非商业的），并且使用SQL来查询我们所需要的信息。如果你不熟悉SQL，我建议你阅读本节前快速浏览附录D。

数据库的填充可以通过类似下面的代码来完成。该例子使用SQLite作为数据库后端，但把它换成其他的数据库（如MySQL、Microsoft SQL Server、Oracle、DB2等）后端也很容易。我们只需要修改DBI连接字符串并确保某个表是否存在（被创建）的代码就好了。下面详细了解一下：

```
use DBI;
use Sys::Hostname;
use strict;

my $db      = 'lastdata';
my $table   = 'lastinfo';

# 该表中我们需要用到的字段名
my @fields = qw( username localhost otherhost whenl );

my $lastex = '/bin/last' if ( -x '/bin/last' );
$lastex = '/usr/ucb/last' if ( -x '/usr/ucb/last' );

# 数据库特定的代码（注意：用户名/密码为空，这个不常见）
# 使用了 RaiseError，这样我们就不必对每个操作都检查是否执行成功
```

```

my $dbh = DBI->connect(
    'dbi:SQLite:dbname=$db.sql3',
    '', '',
    {
        PrintError      => 0,
        RaiseError      => 1,
        ShowErrorStatement => 1,
    }
);

# 检查当前数据库中的表名。
# 这段代码和数据库引擎稍微有些特定的关系，因为这里用到了 map() 来去除由
# DBD::SQLite 返回的表名中所包含的引号。大部分的数据库引擎不需要做这样的
# 操作，因此 $dbh->tables() 返回的值可以被直接使用。
my %dbtables;
@dbtables{ map { /\\"(.*)\\"/, $1 } $dbh->tables() } = ();

if ( !exists $dbtables{$table} ) {

# 另外一些数据库引擎特定的代码。
# 创建所有字段类型为 text 的表。如果使用其他数据库引擎，我们可能
# 会相应地使用 char 和 varchar。
    $dbh->do(
        "CREATE TABLE $table ( " . join( ' text, ', @fields ) . ' text' );
    }

my $thishost = &hostname;

# 构建并准备一个包含占位符的 SQL 语句，即：
# "INSERT INTO lastinfo(username,localhost,otherhost,whenl)
#   VALUES (?, ?, ?, ?)"
my $sth = $dbh->prepare( "INSERT INTO $table ( "
    . join( ' ', @fields )
    . ') VALUES ( '
    . join( ' ', ('?') x @fields )
    . ')' );

open my $LAST, '-|', "$lastex" or die "Can't run the program $lastex:!\n";

my ( $user, $tty, $host, $day, $mon, $date, $time, $whenl );
my ( %users, %connects );
while ( defined( $_ = <$LAST> ) ) {
    next if /^reboot/ or /^shutdown/ or /^ftp/ or /^account/ or /^wtmp/;
    ( $user, $tty, $host, $day, $mon, $date, $time ) = split;
    next if $tty =~ /^:O/ or $tty =~ /^console$/;
    next if ( length($host) < 4 );
    $whenl = $mon . ' ' . $date . ' ' . $time;

# 实际将数据插入到数据库中去。
    $sth->execute( $user, $thishost, $host, $whenl );
}

close $LAST;
$dbh->disconnect;

```

这段代码创建了表lastinfo，包含字段username、localhost、otherhost及whenl。我们遍历last的输出，将所有真实的记录插入到该表中。

现在我们可以用数据库来完成它们所擅长的工作了。下面是一些SQL查询的例子，这些例子可以很容易在Perl中通过第7章中探索过的DBI或者ODBC接口封装起来：

```
-- how many entries in the database?
select count (*) from lastinfo;

-----
      10068

-- how many users have logged in?
select count (distinct username) from lastinfo;

-----
       237

-- how many separate hosts have connected to our machines?
select count (distinct otherhost) from lastinfo;

-----
      1000

-- which local hosts has the user "dnb" logged into?
select distinct localhost from lastinfo where username = "dnb";
localhost
-----
host1
host2
```

当所有的数据都在真正的数据库中时，我们就可以尝试一下所谓的“数据挖掘”了。以上的各个查询请求只要花费1秒左右。数据库可以成为系统管理员快速、强力的工具。

创建自己的日志文件

我有意将关于如何创建你自己的日志文件放到本章最后来讨论，原因很简单：当你了解了如何读取、解析和分析随机日志文件后，才会知道该如何编写代码来生成易读、易解析和易分析的日志文件。正如你接下来要看到的一样，创建日志文件的原理其实很简单，但知道如何去创建好的有用的日志文件确实是一门深厚的艺术。

Perl有数量相对较多的模块可以用来帮助生成日志文件。为了节约篇幅，我们将通过三种解决方案来向你展示这些宝贵的模块所能提供的不同级别的功能性和复杂性。

注意：在我们介绍这些模块前有一点需要了解：你并不一定需要使用任何模块来创建日志文件，实际上这个过程可以很简单：

```
open my $LOGFILE, '>>', 'logfile' or
    die "can't open logfile for append: $!\n";
print $LOGFILE 'began logfile example: ' .
    scalar localtime . "\n";
close $LOGFILE;
```

但如你所想，我们需要的功能可能比这个要更好一点……

日志记录的快捷方式和格式化帮助

首先要提到的是使用日志模块，它能让生成日志行变得更便捷，或具有更好的结构，或同时满足这两个目标。举例来说，Chris Reinhardt的Tie::LogFile模块就能让我们更便捷地创建带预定格式的日志记录行。下面是一段代码范例：

```
use Tie::LogFile;

tie( *LOG, 'Tie::LogFile', 'filename', format => '(%p) [%d] %m' );

print LOG 'message';    # 输出类似: (pid) [dt] message

close(LOG);
```

tie()这一行创建了一个绑定了指定属性的文件句柄。每当我们输出到该句柄，它都会将内容按指定的格式字符串进行格式化并将内容添加到文件中。在本例中，我们制定了每行包含：

(%p)

当前运行进程的PID

[%d]

日期/时间戳

%m

实际的消息（本例中是“message”）

如果我们运行前面这个程序三次，我们将得到类似这样的文件：

```
(19064) [Wed Jun 21 12:01:46 2008] message
(19719) [Wed Jun 21 12:09:02 2008] message
(19725) [Wed Jun 21 12:10:12 2008] message
```

初级/中级日志记录框架

总有一天，我们会对目前介绍过的这些模块不满意，进而寻求更多高级的日志记录功能。此时你会发现自己在寻找一个模块能提供基本的处理日志记录任务的框架。Perl社区中有两个这样的模块比较受欢迎，一个是Dave Rolsky的Log::Dispatch模块，另外一个是最初由Raphael Manfredi开发，现在由Mark Rogaski维护的Log::Agent模块。我们将介绍第一个模块，但你可以自己比较一下这两个模块，看看哪个模块更适合你。

下面介绍Log::Dispatch模块是如何工作的。首先，你需要创建一个日志分派对象，所有的日志记录工作都是通过它来完成的：

```
use Log::Dispatch;
my $ld = Log::Dispatch->new;
```

这个对象本身并不是直接用来记录日志的，它扮演的是总管的角色（这也正是它有趣的地方），它旗下会指定其他一系列的模块来处理每条日志消息。举例来说，如果你想将日志消息记录到一个文件，你可以将如下代码添加到日志分派对象中去：

```
$ld->add(
    Log::Dispatch::File->new(
        name      => 'to_file',
        filename  => 'filename',
        min_level => 'info',
        max_level => 'alert',
        mode      => 'append'
    )
);
```

这段代码指定了输出内容应该传到名为to_file的对象，然后由它将日志数据写到文件filename中去。这个对象会将它接收到的任何级别范围满足info到alert的消息记录下来。

为什么只满足于将日志记录到文件呢？不如我们配置一下要求它再将消息通过E-mail发送出去？你可以这么做：

```
$ld->add(
    Log::Dispatch::Email::MailSend->new(
        name      => 'to_email',
        min_level => 'alert',
        to        => [qw ( operators@example.com )],
        subject   => 'log alert'
    )
);
```

同样地，你也许想将消息发送到syslog服务器以便于之后的集中化和处理：

```
$ld->add(
  Log::Dispatch::Syslog->new(
    name      => 'to_syslog',
    min_level => 'warning',
    facility  => 'local2'
  )
);
```

Log::Dispatch模块非常好的一点是它拥有很多类似这样可用的分派对象模块。它本身还自带了能将消息写入文件、发送电子邮件或输出到屏幕的模块。其他人还创建了能将日志通过DBI模块记录到数据库中去的模块、能写入文件并自动轮循（auto-rotation）的模块以及能通过Jabber即时通信服务器（IM）发送消息的模块等。

善于观察的读者可能有些不耐烦了，因为只介绍模块，到目前为止我们还没实际记录过任何东西。没问题，这个简单：

```
$ld->log( level => 'notice', message => 'here is a log message' );
```

或者，我们可以用一个快捷的方法发送一条notice级别的消息：

```
$ld->notice( 'here is a log message' );
```

上面这段代码会将那条消息发送到所有我们通过add()方法添加的满足该记录级别的日志分派对象。如果此时是紧急情况，我们调用了如下方法：

```
$ld->emergency( 'printer on fire!' );
```

那么这条消息会同时记录到文件并发送至syslog，而且还会发送一封电子邮件。若想将一条消息发送到指定的日志分派对象，我们可以使用log_to方法：

```
$ld->log_to( name   => 'to_syslog',
             level  => 'debug',
             message => 'sneebie component is failing' );
```

类似这样的初级/中级日志记录框架显著的优势就是能让我们更有效地控制什么时候记录消息、记录到什么地方。一般情况下，项目所需要的灵活性和可控性都能被满足。但实际上还有些情况，有些更大的项目需要更多的可控性。针对这种情况，我们还有可用的高级日志框架。

高级日志记录框架

接下来这个拥有更强功能性和复杂性的框架是Mike Schilli和Kevin Goess编写的log4perl日志记录框架。这个框架直接移植自Java社区十分流行的log4j框架。log4perl包和原框架高度兼容，它甚至可以不加修改地解析和使用很多log4j的配置文件。

在这里举更多的代码范例或者深入探讨log4perl的功能除了占用更多篇幅外，并不能帮你更好地理解它，何况它的作者本身已经编写了一套很好的教程（可以从这章末尾的参考资料部分找到相关信息）。相反，让我们简明扼要地向你介绍一下这个框架的特性以及它能如何让你受益。

让我们从之前见到过的特性开始：

- log4perl提供和Log::Dispatch模块一样的将日志消息输出到不同目的地的功能。其实它本身也和Log::Dispatch模块一样用的是Log::Dispatch::*系列输出模块，所以这些功能和灵活性它也同样具备。
- log4perl支持日志记录分级（并且有能力通知框架的有关部分只关注满足特定级别的日志消息）。
- log4perl对所有标准级别的日志记录都具有类似的方便易用的方法（如\$object->error()、\$object->warn()、\$object->debug()等）。

下面让我们介绍令人激动的部分：

- log4perl具有被称为“类别”的功能，让你可以为实现日志记录的特定部分代码命名。以一个在线银行的应用程序为例，你可能会有GetBalance、MakeWithdrawal和MakeDeposit这三个类别来分别对应你的代码的相关部分。这些类别的日志记录功能都能够独立地开启和关闭，同时还可以分别设定日志记录级别。只想在debug级别记录withdrawals的日志？使用log4perl完全没问题。
- 如果你正在构建大型的代码量很多的系统，你很可能是通过面向对象的方式编写程序的，包括类和子类、对象和子对象以及其他一些不错的东西。这些复杂性log4perl全都可以处理，因为它的类别本身也是可以继承的。每个类别正好可以对应于你的系统中的相应类，比如你可以有Withdrawal、Withdrawal::CheckBalance、Withdrawal::CheckBalance::Overdraft等这些类。你可以在类别树层级比较高的位置设置日志级别，这样从那个位置起下面的所有子类别都会继承同样的设置。因此你想开启复杂的代码中某块代码的日志记录功能就很容易了。
- 正如我之前提到的那样，log4perl可以读取配置文件，这些配置文件可以精确地描述针对你的复杂代码该如何去开启日志记录功能。更棒的是，log4perl还可以被设置为周期性地检查该配置文件是否被修改，一旦被修改就加载新的配置文件。这意味着可以在你庞大的系统运行的同时对日志记录功能做出修改，非常灵活。

如果这些介绍激起了你的兴趣，你可以访问<http://log4perl.sourceforge.net>以获取更多详细资料。

日志的创建、操作和分析是一个庞大的主题。希望本章的内容能让你了解一些相关的工具，能给你带来一些灵感。

本章所用模块

模块名	CPAN ID	版本
Win32::EventLog (随 ActivePerl 发布)		0.074
File::Copy (随 ActivePerl 发布)		2.09
Logfile::Rotate	PAULG	1.04
File::Temp (随Perl发布)		0.17
Getopt::Long (随Perl发布)		2.35
Time::Local (随Perl发布)		1.13
Perl6::Form	DCONWAY	0.04
User::Utmp	MPIOTR	1.8
Readonly	ROODE	1.03
Log::Procmail	BOOK	0.11
SyslogScan	RHNELSON	0.32
File::Tail	MGRABNAR	0.99.3
Parse::Syslog	DSCHWEI	1.09
Regexp::Log::DateRange	KARASIK	0.01
Regexp::Log	BOOK	0.04
Regexp::Log::Common	BARBIE	0.04
Log::Statistics	VVU	0.047
DB_File (随Perl发布)	PMQS	1.72
DBM::Deep	RKINYON	0.983
FreezeThaw	ILYAZ	0.3
Sys::Hostname (随Perl发布)		1.11
Fcntl (随Perl发布)		1.05
DBI	TIMB	1.52
DBD::Sqlite	MSERGEANT	1.13
Tie::LogFile	CREIN	0.1
Log::Dispatch	DROLSKY	2.13
Log4perl	MSCHILLI	1.07

更多参考资料

《Essential System Administration》(Third Edition)，由Eleen Frisch所著(O'Reilly出版)，其中有对syslog很好的简短介绍。

<http://www.heysoft.de/index.htm>是Frank Heyne软件的主页，它提供了Win32事件日志解析软件。该网站上有不错的关于事件日志的常见问题列表。

<http://www.le-berre.com>是Philippe Le Berre的主页，它包括很多不错的关于使用Win32::EventLog模块及其他Windows模块的文章。

《Practical Unix & Internet Security》(Third Edition)，由Simson Garfinkel、Gene Spafford及Alan Schwartz合著(O'Reilly出版)，是另外一个不错的(且更详细些的)介绍syslog的资料，它还包括tcpwrappers的信息。

<http://www.geekfarm.org/wu/muse/LogStatistics.html>是Log::Statistics模块的主页，而且包含了关于该项目的不错的文档资料。

<http://log4perl.sourceforge.net>是log4perl项目的主页。请务必阅读该站点上链接到<http://www.perl.com/pub/a/2002/09/11/log4perl.html>的教程。

<http://www.loganalysis.org>是Tina Bird和Marcus Ranum所建的站点，他们是两位安全研究员，致力于引起大家对与安全相关的日志分析问题的关注。他们还建立了关于该项目的邮件列表：<http://www.loganalysis.org/mailman/listinfo/loganalysis/>。

USENIX协会在2008年举办了一次系统日志分析的研讨会(即WASL'08)。更多的信息可以在这里找到：<http://www.usenix.org/events/wasl08/>。

关于交互式日志分析，由Splunk提供的产品(<http://www.splunk.com>)是其中的佼佼者。如果需要分析的数据在指定大小范围以内，他们的产品还允许你免费使用。

微软做过一个叫做Log Parser的包(尽管缺乏宣传，但它非常酷)。我最近一次是在<http://www.microsoft.com/downloads/details.aspx?FamilyID=890cd06b-abf8-4c25-91b2-f8d975cf8c07>这个下载站找到它的，但鉴于微软老是变换链接地址，所以你也许得去<http://www.microsoft.com/downloads/搜索该包>。微软是这样描述它的：

Log Parser是个强大的多功能工具。对于类似日志文件、XML文件及CSV文件等这些基于文本的数据，以及Windows®操作系统中类似事件日志、注册表、文件系统和活动目录(Active Directory®)等这些关键数据源，Log Parser提供了统一的查

询访问功能，你只需要告诉Log Parser需要什么信息以及希望这些信息如何被处理即可。可以自定义查询返回结果的格式，以文本方式输出，或者以其他更专门的格式输出，如SQL、SYSLOG或图表等。

任何关于安全问题的讨论都有风险，原因有三：

- 对于不同的人来说安全有不同的含义。如果你参加一个古希腊罗马行政官会议，并且问到关于罗马的信息，那么第一位行政官会告诉你关于饮用水的渠道架设的情况，而第二位则会专注于“罗马和平”（理念和政策），第三位会介绍罗马士兵的行军方阵，第四位则力图解释罗马议会的管理机制，等等。安全这个词背后的各种含义也是同样无法一概而论。
- 安全是一个连续体，而不是非黑即白。人们常常误认为一个程序、一台电脑或者一个网络本身可以成为安全的。这一章不能帮你把某个东西安全化，不过会帮你使得某个东西更加安全，或者最起码能让你认识到一些不够安全的东西。
- 最后一个常见的误区是为安全制定规范。虽然你可以通过规范来关注一些细节因素，但是总是有新的问题需要关注。比如说给A、B、C三个漏洞打上补丁只能（在补丁正确工作的情况下）避免这三个漏洞导致的问题，对于发现D漏洞并不会有帮助。这就是为什么这一章会关注安全问题的基本原则和工具，而不是教你如何修补某个缓冲溢出、易损的注册表项或者某个人人都能修改的系统文件。

讨论安全问题时一个很好的起点就是考察现实世界的安全话题。你会发现无论在虚拟世界还是现实世界中，所有的安全问题都会导致担心。我关注的某个东西是否会被毁坏、丢失或者显现出来？我能做一些事情来避免这些糟糕的事情发生吗？这些事情发生的可能性有多大？发生后会导致多大的影响？现在有没有发生呢？

如果我们能在现实世界实际面对这些值得担心的事情，那么在系统管理领域也应该能妥善处理。在需要保护现实世界的物品时，我们发明了特别的空间隔离机制（比如银行保险箱），这样只有某些人可以操作它们。在需要保护现实世界的知识产权和机密的时候，我们创造了限制访问的方法，如绝密许可政策以及针对间谍的数据加密技术。对计算机来说很容易找到对应的技术，比如权限系统、访问列表、加密等。不过无论对于哪

个世界，安全永远都是一种永无止境的追求而已。对于为安全系统的设计而花费的每一分钟，都能发现尝试破解它的另一分钟。对于我们来说，威胁不仅来自于那些不安分的孩子（他们只要一上网就想找到点惊喜），还来自于那些想找机会报复的离职员工。

多年以来，尝试解决安全问题的常见方法就是指定一个人来平息公众的担心。一段时间以来，守望者的脚步声总能让人感觉到安全。下面我们会尝试让Perl成为系统和网络安全守望者，守住安全问题的大门。

注意不必要的或未授权的修改

一个好的守望者会关注变化。他会发现某些不对劲的事情，或者突然消失的东西。如果名贵的马耳他之鹰被换成了赝品，那么守望者应该是首先注意到的人。同样地，如果有人修改（或替换）了系统的重要文件，那么你应该希望在第一时间收到警报。虽然大多数情况下的修改都是无害的，但是如果有人擅自修改了/bin/login、system32/*.dll或者Finder，那么你会觉得之前收到的那些误报都是值得的。^[注1]

本地文件系统的修改

文件系统是绝佳的演示变动监控程序的场合。我们会试着捕捉重要文件的变动，比如/etc/passwd、system32/*.dll是否被修改。在不告知系统管理员的情况下修改这些文件往往意味着系统入侵。某些复杂的网络蠕虫能悄悄覆盖重要的系统文件，并且会掩盖修改的痕迹。这种类型的恶意修改我们应该能捕获。^[注2]另一方面，即使没有入侵，能够得知这些重要的文件何时被修改也是有益的（尤其是在多人管理同一个系统的时候）。我们将要介绍的技术能对这两种情况奏效。

最简单的识别文件修改的方法是使用Perl函数stat()和lstat()。这些函数能接受文件名或者文件句柄，返回关于文件的信息列表。这两个函数的区别在于对类Unix系统的符号链接的处理。在遇到符号链接的时候，lstat()返回的是链接本身的信息，而stat()则会返回被链接文件的信息。在所有其他操作系统上，lstat()会返回和stat()相同的信息。

使用stat()或者lstat()非常容易：

```
my @information = stat('filename');
```

注1： 这并不意味着你不应该努力避免错误警报。如果收到太多的误报，那么你会自然开始忽略它们，有时甚至会把它们自动发送到垃圾箱。

注2： 不过如果某个极其狡猾的rootkit替换了Perl调用的操作系统级函数，那么对不起，所有的努力都是枉费。

如同在第2章展现的，我们也可以通过Tom Christiansen的File::Stat模块来获得使用面向对象语法的信息。

stat()和lstat()返回的信息是与平台相关的。stat()和lstat()最开始是作为Unix系统调用，所以Perl说明文档是按照Unix系统来介绍它们的。表11-1展示了stat()在基于Windows的操作系统上的返回值与Unix系统返回值的对比。前两列展示了Unix平台的字段编号和信息描述。

表11-1: stat()返回值对比^a

字段编号	Unix字段信息描述	字段对基于Windows的操作系统有效吗?
0	文件系统设备编号	是 (分区号)
1	Inode编号	否 (总是返回0)
2	文件模式 (类型和权限)	是
3	文件的 (硬) 链接计数	是 (针对NTFS)
4	文件所有者的 (数字) 用户ID	否 (总是返回0)
5	文件所有者的 (数字) 组ID	否 (总是返回0)
6	设备标识符 (针对特殊文件)	是 (分区号)
7	文件大小 (按字节计算)	是, 但不包括NTFS交换数据流的大小
8	最近访问时间 (以epoch为起点)	是
9	最近修改时间 (以epoch为起点)	是
10	Inode修改时间 (以epoch为起点)	是 (但其实是文件创建时间)
11	文件系统I/O的最佳块大小	否 (总是空)
12	实际分配的块数	否 (总是空)

a. 本书第一版的读者可能注意到这个表格少了一列，这是因为Mac OS到Mac OS X的改进提高了与Unix的兼容度，所以老的Mac OS列就不再需要了。

警告: 如果对你来说，在Windows系统上以Unix的方式来处理时间值非常重要，那么可以安装Steve Hay的Win32::UTCFileTime模块，并且仔细阅读相应的文档。Windows系统在处理文件时间的夏时制方面有些问题，不过这个模块能覆盖Perl标准的stat()调用（同时也覆盖了一些其他调用），从而修正了时间计算问题。

除了stat()和lstat()以外，还有一些和操作系统相关的特殊Perl模块能返回文件的特殊属性。请参考第2章介绍的那些函数，比如Win32::FileSecurity::Get()。

一旦获得了某个文件的stat()值，就可以和之前产生的版本进行比较，从而了解是否有变动。如果这些值发生了改变，那么文件就可能经历了某种修改。下面的程序能产生

lstat()返回值的列表，也能和之前保存的值进行对比。我们在此特意跳过了表11-1中的第8个字段（最近访问时间），因为它每次文件被读取之后都会改变。

这个程序能接受-p *filename*参数来打印某个文件的lstat()值，还能接受-c *filename*参数来检查*filename*中列出的所有文件的stat()值：

```
use Getopt::Std;

# 稍后我们将用 PrintChanged() 以漂亮的形式输出此处的信息
my @statnames = qw(dev ino mode nlink uid gid rdev
    size mtime ctime blksize blocks);

getopt( 'p:c:', \my %opt );

die "Usage: $0 [-p <filename>|-c <filename>]\n"
    unless ( $opt{p} or $opt{c} );

if ( $opt{p} ) {
    die "Unable to stat file $opt{p}:$!\n"
        unless ( -e $opt{p} );
    print $opt{p}, '|', join( '|', ( lstat( $opt{p} ) )[ 0 .. 7, 9 .. 12 ] ),
        "\n";
    exit;
}

if ( $opt{c} ) {
    open my $CFILE, '<', $opt{c}
        or die "Unable to open check file $opt{c}:$!\n";
    while ( <$CFILE> ) {
        chomp;
        my @savedstats = split('\|');
        die "Wrong number of fields in line beginning with \"$savedstats[0]\n"
            unless ( scalar @savedstats == 13 );
        my @currentstats = ( lstat( $savedstats[0] ) )[ 0 .. 7, 9 .. 12 ];

        # 如果有数据改动，则在此处打印这些改动过的字段
        PrintChanged( \@savedstats, \@currentstats )
            if ( "@savedstats[1..12]" ne "@currentstats" );
    }
    close $CFILE;
}

# 迭代属性列表，如果有数据改动，则打印前后内容
sub PrintChanged {
    my ( $saved, $current ) = @_;

    # 从预备列表中取出要检查的文件，并打印该文件的名字
    print shift @{$saved}, ":\n";

    for ( my $i = 0; $i <= $#{$saved}; $i++ ) {
        if ( $saved->[$i] ne $current->[$i] ) {
            print "\t" . $statnames[$i] . ' is now ' . $current->[$i];
            print ' (should be ' . $saved->[$i] . ")\n";
        }
    }
}
```



```
}
}
}
```

使用这个程序的时候，我们可以键入`checkfile -p /etc/passwd >> checksumfile`。这样`checksumfile`就会有这样一行：

```
/etc/passwd|1792|11427|33060|1|0|0|24959|607|921016509|921016509|8192|2
```

我们可以对需要监控的其他文件继续执行这一行，然后使用`checkfile -c checksumfile`来显示变动。比如只要删除`/etc/passwd`文件中的一个字符，这个脚本就会这样汇报：

```
/etc/passwd:
size is now 606 (should be 607)
mtime is now 921020731 (should be 921016509)
ctime is now 921020731 (should be 921016509)
```

在我们转换话题之前，有必要分享一个Perl技巧。下面这一行代码展示了快速比较两个列表内容的技巧：

```
if ("@savedstats[1..12]" ne "@currentstats");
```

Perl能自动把这两个列表转化成字符串（通过在列表元素中加入空格），可以理解为下面的代码：

```
join(' ',@savedstats[1..12]))
```

然后再对两个结果字符串进行比较。这个技巧对那些元素顺序固定的列表能奏效，不过列表元素中必须避免出现列分隔符（也就是`$`，通常是空格）。如果这个技巧不奏效，请考虑`Array::Compare`这样的模块。

现在你已经掌握了文件属性，不过还不足够。虽然对文件属性的监视是很好的开始，不过这远远不够，因为修改文件之后再把它属性恢复成修改之前的样子并不难。Perl里面就有一个叫做`utime()`的函数能修改文件的属性（包括最近修改时间）。现在看来有必要使用更加厉害的武器。

监视数据修改是一种叫做“消息摘要算法”的算法特长。下面摘录的是Ron Rivest在RFC 1321中对“RSA Data Security, Inc. MD5 Message-Digest Algorithm”的描述：

这个算法能接受任意长度的信息，输出相应的128位的“指纹”（或“消息摘要”）。普遍认为此算法不可能对两段不同的消息产生同样的消息摘要，也不能根据消息摘要推算出原始消息的内容。

对我们来说，这意味着可以用MD5（或进一步使用SHA）这样的消息摘要算法对某个文

件算出唯一的指纹。只要某个文件被修改了，不论多小的改变，都会导致文件的指纹产生变化。

MD5现在被认为有害了么？

这里要对Dan Kaminsky表示歉意，他是我喜欢的安全研究员。在本章末尾的参考资料中列出了他的论文《MD5 To Be Considered Harmful Someday》，这里要指出为什么本书上一版的MD5代码都消失了。

自从本书的第一版上市之后，Ron Rivest关于“此算法不可能对两段不同的消息产生同样的消息摘要”的断言已经显得有些理想化了，目前有越来越多的数学领域和计算机领域的证据表明这种说法并不正确。现在可以使用强大的计算阵列或使用冗余计算功能作为辅助，所以找到具有相同MD5消息摘要的两个不同文件变得更加容易了。

注意我说的是“更加容易”，而不是“容易”。这段补充内容可能也会在将来显得理想化，不过我还是要大胆推断，如果有人正在企图把你的某个重要文件替换为具有相同MD5的其他文件，那么你可能会面对比消息摘要算法更大的麻烦。

可以这么说，把我的范例程序从MD5改成SHA-256只需要花5秒钟，因为Mark Shelor的Digest::SHA模块和Gisle Aas的Digest::MD5模块有几乎相同的接口。另外它还能支持SHA-2算法，所以你可以根据需要切换成更加长的信息摘要。现在你看我们已经摆脱了MD5算法，所以请不要继续嘲笑我们了。

对Perl来说最简单的方法是使用Digest系列模块中的Digest::SHA模块。

Digest::SHA模块很容易使用。只要先创建一个Digest::SHA对象，然后通过add()或者addfile()方法来载入数据，最后请求模块创建摘要。

要使用SHA-256算法来给Unix密码文件计算摘要，可以使用下面的代码：

```
use Digest::SHA;

my $sha = Digest::SHA->new(256);

# 'p' 表示 'portable mode'，它会将换行符转换为 Unix 格式的换行符，
# 以对相同内容求得一致的摘要，这对兼容不同系统的数据很有用。
# 如果觉得心里不踏实，尽管去掉吧。
$sha->addfile( '/etc/passwd', 'p' );

print $sha->hexdigest . "\n";
```

你还可以把方法调用串联起来，这样程序会更加精简：

```
use Digest::SHA;

print Digest::SHA->new(256)->addfile( '/etc/passwd', 'p' )->hexdigest, "\n";
```

以上代码都能打印同样的输出：

```
c0e541600943622fe8ddf4142072107f076a8da35d1e39bc1c8c91a3892a46da
```

如果我们对文件做了细微修改，输出就会有显著变化。下面的输出是对调了密码文件里面的两个字符之后产生的：

```
ef88f8ce4c24eaa2d5937e929955d0eb63caf4813026ca8c877e3cc4b123c3ac
```

任何的数据修改都变得很显著。如果我们把修改撤回，消息摘要会恢复原样，不过 `stat()` 仍然会显示文件已经被修改过了。

下面就尝试把SHA-256算法集成到刚才的属性检查程序中：

```
use Getopt::Std;
use Digest::SHA;

# 稍后我们将用 PrintChanged() 以漂亮的形式输出此处的信息
my @statnames = qw(dev ino mode nlink uid gid rdev
    size mtime ctime blksize blocks SHA-256);

getopt( 'p:c:', \my %opt );

die "Usage: $0 [-p <filename>|-c <filename>]\n"
    unless ( $opt{p} or $opt{c} );

if ( $opt{p} ) {
    die "Unable to stat file $opt{p}:$!\n"
        unless ( -e $opt{p} );

    my $digest = Digest::SHA->new(256)->addfile( $opt{p}, 'p' )->hexdigest;

    print $opt{p}, '|', join( '|', ( lstat( $opt{p} ) )[ 0 .. 7, 9 .. 12 ] ),
        "|$digest\n";
    exit;
}

if ( $opt{c} ) {
    open my $CFILE, '<', $opt{c}
        or die "Unable to open check file $opt{c}:$!\n";
    while (<$CFILE>) {
        chomp;
        my @savedstats = split('\|');
        die "Wrong number of fields in line beginning with $savedstats[0]\n"
            unless ( scalar @savedstats == 14 );
        my @currentstats = ( lstat( $savedstats[0] ) )[ 0 .. 7, 9 .. 12 ];
```

```

        push( @currentstats,
              Digest::SHA->new(256)->addfile( $savedstats[0] )->hexdigest );

        # 如果有数据改动，则在此处打印这些改动过的字段
        PrintChanged( \@savedstats, \@currentstats )
            if ( "@savedstats[1..13]" ne "@currentstats" );
    }
    close $CFILE;
}

# 迭代属性列表，如果有数据改动，则打印前后内容
sub PrintChanged {
    my ( $saved, $current ) = @_ ;

    # 从预备列表中取出要检查的文件，并打印该文件的名字
    print shift @{$saved}, ":\n";

    for ( my $i = 0; $i <= $#{$saved}; $i++ ) {
        if ( $saved->[$i] ne $current->[$i] ) {
            print "\t" . $statnames[$i] . ' is now ' . $current->[$i];
            print " (should be " . $saved->[$i] . ")\n";
        }
    }
}

```

最后一个关于文件系统监控的技巧是：使用操作系统内置的文件系统修改捕获机制。Linux可以使用`inotify`（也就是以前的`dnotify`）；Mac OS X 10.5之后的版本可以使用文件系统事件API（Andy Grundman的`Mac::FSEvents`模块为Perl程序铺平了道路）；Windows可以使用内置的审核机制，这在第4章中介绍过。相信其中必然有一种工具对你来说是可用的。

网络数据的改变

我们已经介绍了如何捕获本地文件系统的修改。那么如何捕获其他机器（或者外部服务）的数据修改呢？在第5章中我们展示了查询NIS和DNS的技巧，其实也很容易通过定时的查询来发现数据的更新。比如对于DNS服务器来说，只要配置允许，我们就可以伪装成从服务器并通过“zone transfer”来转储服务器上某个区域的数据：

```

use Net::DNS;

# 从命令行获取两个参数：第一个是要查询的域名服务器，
# 第二个是要用该服务器查询的域名
my $server = new Net::DNS::Resolver;
$server->nameservers( $ARGV[0] );

print STDERR 'Transfer in progress...';
my @zone = $server->axfr( $ARGV[1] );
die $server->errorstring unless @zone;
print STDERR "done.\n";

```

```
foreach my $record (@zone) {
    $record->print;
}
```

注意：所有配置正确的DNS服务器都会限制执行zone transfer（区域传送）的客户机。所以这样的程序必须在授权的主机上才能正确运行。

现在让我们把这个想法和SHA-256结合。所以我们不再打印区域信息，转而计算它的摘要：

```
use Net::DNS;
use FreezeThaw qw(freeze);
use Digest::SHA;

my $server = new Net::DNS::Resolver;
$server->nameservers( $ARGV[0] );

print STDERR 'Transfer in progress...';
my @zone = $server->axfr( $ARGV[1] );
die $server->errorstring unless @zone;
print STDERR "done.\n";

my $zone = join( '', sort map { freeze($_) } @zone );

print "SHA-2 fingerprint for this zone transfer is: \n";
print Digest::SHA->new(256)->add($zone)->hexdigest, "\n";
```

SHA-256和其他消息摘要算法一样，只能对单个标量数据块进行计算，所以不能把@zone这样的Perl哈希列表数据结构传递给它。所以下面的这行代码成为必需的：

```
my $zone = join( '', sort map { freeze($_) } @zone );
```

我们使用了在第10章介绍过的FreezeThaw模块来把每个@zone记录数据结构转化成纯文本字符串。任何其他相似的模块也可以达到同样的效果，比如Data::Dumper。转化完成后，字符串经过排序后拼接成更大的标量值。这里的排序使得区域传送的顺序不会影响结果。

转储整个区域文件的内容这个想法有点极端，尤其是对于较大的区域传送域来说，所以可以把这个监控范围限制在某些重要的子网或几个重要地址。另外，限制区域传送的客户机也非常有必要，这对安全有益。

这里列出的思路并不会把你完全带出困境。下面的几个问题你可能会遇到：

- 如果有人企图修改你的SHA-256摘要数据怎么办？万一其中的摘要和原始数据一起被改掉呢？

- 如果有人企图篡改你的脚本怎么办？如果脚本看上去在计算摘要，而其实没有呢？
- 如果有人篡改了你的系统上的SHA模块怎么办？
- 对于那些怀疑到底的人来说，如果有人篡改了Perl可执行程序或者它的共享库怎么办？万一修改的是操作系统内核呢？^[注3]

通常这样的威胁只能通过使用只读介质进行备份来避免。可以对任何可能被篡改的东西进行备份，摘要数据、模块、Perl静态编译的可执行程序等等。

这个问题反映了安全问题的另一面：如果仔细推敲，就会发现几乎所有的东西都不安全。所以问题在于如何找到一个鼓励怀疑和默许懒惰的平衡点。

关注可疑行为

一个好的守望者不只是擅长发现变动，他还需要关注可疑的行为和状况。有人需要注意铁丝网上的洞以及夜里发出的奇怪撞击声。我们的程序应该也可以完成这个任务。

本地的问题信号

可惜的是，对可疑行为的敏感往往来自于痛苦的经历，这样才能真正激发避免它的动力。在经历了几次安全方面的攻击之后，你会发现入侵者往往有固定的入侵模式，也会留下一些隐蔽的痕迹。一旦你知道了这些迹象，使用Perl来显示入侵就比较容易了。

注意： 在每次安全攻击之后，很有必要花点时间写一个书面的分析报告。详细记录入侵发生的位置、他们使用的工具（或漏洞）、他们入侵之后的动作、是否还攻击了其他系统、你做了什么回应等等。

往往很难避免的就是恢复到正常的生活，彻底忘记这次不愉快的经历。如果你不能忘记它，那么你会发现自己已经学到了一些东西，这样就能避免问题重复发生。尼采所说的“那些没有消灭你的东西，会使你变得更强壮”对系统管理来说还是有些道理的。

比如入侵者（尤其是那些比较简单的）往往试图通过隐藏目录的方法来存储自己的数据。在Unix系统上它们往往试图把入侵代码和嗅探器输出放在类似“...”（三个点号）、“.”（点号空格）或者“ Mail ”（空格Mail）这样的目录中。因为这样的目录往往难以通过`ls`发现。

注3： 如果你还没有读过Ken Thompson的论文《Reflections on Trusting Trust》，那么现在就是阅读的时候了。

你可以使用第2章介绍的工具轻易地编写这种目录的搜索程序。下面的程序就可以使用 `File::Find` 模块来搜索名字罕见的目录：

```
use File::Find;

find( \&wanted, '.' );

sub wanted {

    ( -d $_ ) and      # 只关心目录
    $_ ne '.' and $_ ne '..' and      # 并且不是 . 或者 ..
    (
        /^[^-.a-zA-Z0-9+,:;_~\${}()]/ or      # 包含一个“使坏”的字符
        /\^\.{3,}/ or      # 或者以三个点号开头
        /\^-/ or      # 或者以破折号开头
    ) and print "' ' . nice($File::Find::name) . "'\n";
}

# 打印目录名字的“漂亮”版本，即将控制字符也回显出来。
# 该子例程是从Perl的 dumpvar.pl 中的 &unctrl() 原样截取而来的。
# 如果不想做抄袭者，可以改用 Devel::Dumpvar 这样的模块。
sub nice {
    my $name = shift;
    $name =~ s/([\001-\037\177])/'^'.pack('c',ord($1)^64)/eg;

    return $name;
}
```

更好的方法是使用 `File::Find::Rule` 来编写相同功能的代码：

```
use File::Find::Rule;

my @problems
    = File::Find::Rule->name( qr/^[^-.a-zA-Z0-9+,:;_~\${}()]/,
                             qr/\^\.{3,}/,
                             qr/\^-/ )
    ->in('.');

foreach my $name (@problems) {
    print "' ' . nice($name) . "'\n";
}

# 打印目录名字的“漂亮”版本，即将控制字符也回显出来。
# 该子例程是从Perl的 dumpvar.pl 中的 &unctrl() 原样截取而来的。
# 如果不想做抄袭者，可以改用 Devel::Dumpvar 这样的模块。
sub nice {
    my $name = shift;
    $name =~ s/([\001-\037\177])/'^'.pack('c',ord($1)^64)/eg;

    return $name;
}
```

文件系统筛查程序的有效性往往取决于正则表达式的质量和数量。如果使用了太少的正则表达式，那么可能错过需要发现的问题；如果使用了太多的正则表达式，或者正则表达式过于复杂，程序会运行很久并且使用太多资源。如果你的正则表达式太宽松，程序可能产生大量的误报。所以找到合适的平衡点非常重要。

发现问题模式

我们刚刚讨论了如何搜索可疑的对象，现在我们要进一步发现那些可疑的行为模式（pattern）。我们会展示一个日志文件的分析程序，从而识别可能的入侵。

这个例子会做如下假设：大多数用户会从固定的几个位置登录系统。也就是说，他们通常每次都是从同一台机器或者同一个ISP登录系统。如果你发现某个账户多次从非经常登录的位置登录，那么这往往意味着账户已经被攻破，密码已经被多人知道。当然这个假设对于那些高度可移动的用户（比如那些使用VPN或者公司代理服务器的用户）来说并不成立，不过如果你发现某个用户同时从巴西和芬兰登录，那么这往往意味着你发现了安全问题。

让我们展示这个程序的代码。这里要说明的是，代码依赖于Unix环境，不过它里面展示的技术是跨平台的。首先，让我们先写好内置的说明文档。把说明文档写在程序开始的位置是很好的主意，因为这样有兴趣读代码的人就能首先看到它^[注4]。在往下看之前，请先仔细看看这个程序会支持的参数：

```
sub usage {
    print <<"EOU";
lastcheck - check the output of the last command on a machine
             to determine if any user has logged in from > N domains
             (inspired by an idea from Daniel Rinehart)

USAGE: lastcheck [args], where args can be any of:
-i <class>    for IP #'s, treat class <B|C> subnets as the same "domain"
-f <domain>   count only foreign domains, specify home domain
-l <command>  use <command> instead of default /usr/bin/last -a
              note: no output format checking is done!
-m <#>       max number of unique domains allowed, default 3
-u <user>     perform check for only this username

-h           this help message

EOU
exit;
}
```

注4： 如果你希望给用户留下深刻的印象，还可以使用像Pod::Usage模块的pod2usage()这样的函数来显示脚本内嵌的POD格式的手册页。

我们首先解析用户的命令行参数。下面展示的`getopts`行能够把命令行参数的情况写入`$opt{ <flag letter> :}`。字母后面的冒号代表这个选项需要提供参数值：

```
use Getopt::Std;
use Regexp::Common qw(net);

getopts( 'i:hf:l:m:u:', \my %opt );    # 解析用户输入

usage() if ( defined $opt{h} );

# 唯一域名的数量（默认为 3），超过我们就要发牢骚了
my $maxdomains = $opt{m} ||= 3;

# 子网类型的名字始终保持大写，如果没有指定就默认使用 'C' 网
if ( exists $opt{i} ) {
    $opt{i} = uc $opt{i};
    $opt{i} ||= 'C';
}
```

下面这行反映了我们在第4章讨论的可移植性和效率的权衡。如果你决定提高程序的效率（同时降低可移植性），可以使用在那一章介绍过的`unpack()`。这里我们决定调用外部程序：

```
my $lastex = $opt{l} ||= '/usr/bin/last -a';

open my $LAST, '-|', $lastex || die "Can't run the program $lastex:$!\n";
```

在我们进一步深入介绍程序之前，先看看这个程序使用的哈希的哈希数据结构，它的数据是从`last`命令获取的。在第一级的哈希中，用户名是键，而对应值是一个子哈希。子哈希的键是用户登录的域名，而它的值则无关紧要。我们使用子哈希（而非子列表）是因为这种数据结构更容易保证域名的唯一性。

比如下面这段数据结构：

```
$userinfo { 'laf' } = { 'ccs.example.edu' => undef,
                       'xerox.com'       => undef,
                       'tpu.edu'        => undef }
```

这个条目显示用户`laf`先后从`ccs.example.edu`、`xerox.com`和`tpu.edu`三个域登录过。

我们从逐行扫描`last`命令的输出开始。在我的系统上，输出信息如下：

```
cindy    pts/145  Thu Jan  1 20:57   still logged in  nwbdfsd42.hsd1.ma.comcast.net
michael  pts/145  Thu Jan  1 20:27 - 20:27 (00:00) pool-68-25-87.bos.verizon.net
david    pts/113  Thu Jan  1 18:51   still logged in  65.64.24.204
deborah  pts/110  Thu Jan  1 14:48 - 15:42 (00:54) nat-service4.example.net
barbara  pts/158  Thu Jan  1 10:25 - 11:22 (00:57) 65.96.246.34
jerry    pts/81   Thu Jan  1 10:04 - 12:13 (02:09) atheds1-4392.home.otenet.gr
```

早在while循环中，我们就已经尝试剔除不感兴趣的数据。通常这是一个好的习惯，尽可能早地在循环开始处就进行数据检查，不必拖到数据处理阶段（例如，通过//进行数据提取的时候）。这使得程序更早跳过那些不必要的行，从而读入下一行：

```
my %userinfo;
while (<$LAST>) {

    # 忽略特殊用户
    next if /^reboot\s|^shutdown\s|^ftp\s/;

    # 如果用 -u 指定了一个特定用户，则跳过所有其他用户
    # 而该用户的名字是保存在 $opt{u} 中的，这是 getopts 帮我们保存的
    next if ( defined $opt{u} && !/^$opt{u}\s/ );

    # 忽略来自 X 控制台的登录
    next if /^:0\s+(:0)?/;

    chomp;    # 去掉末尾的换行符

    # 确定用户名字、终端 tty 以及远程主机名
    my ( $user, $host ) = /^( [a-z0-9-\.]+ )\s.*\s( [a-zA-Z0-9-\.]+ )$/;

    # 如果日志记录的用户名不正确，就忽略
    next if ( length($user) < 2 );

    # 忽略名字中没有域名或 IP 信息的记录
    next if $host !~ /\. /;

    # 提取主机名中的域名部分（参考后续代码说明）
    my $dn = domain($host);

    # 如果域名不正常，也忽略
    next if ( length($dn) < 2 );

    # 如果使用 -f 开关指定强制外部域名，则忽略指定域名以外的记录
    next if ( defined $opt{f} && ( $dn =~ /^$opt{f}/ ) );

    # 存储该用户的信息
    $userinfo{$user}{$dn} = undef;
}
close $LAST;
```

这里用到了一个工具子例程domain()，它接收完全限定域名（Fully Qualified Domain Name, FQDN），也就是主机名加全域名，并且返回那台机器的域名。这个子例程需要一些条件判断，因为并非日志中的所有的主机名都是实际的名字，有时只是简单的IP地址。针对这种情况，我们设计了-i开关。如果用户运行脚本时设置了这个开关，那么我们假设所有的IP地址都是B或C类地址。也就是说，我们把IP地址的前两个字节（或前三个字节）当作这台主机的域名。这让我们能把192.168.1.10理解成来自和192.168.1.12相同的域。这并不是最好的假设，不过这是我们在不依赖外界信息的情况下能实现的最好解决方案，而且往往总能工作。如果用户没有使用-i开关，那么我们把整个IP地址当成域。

下面是该子例程的代码，随附快速注释：

```
# 取得 FQDN 并尝试返回 FQD
sub domain {
    my $fdqn_or_ip = shift;

    if ( $fdqn_or_ip =~ /^$RE{net}{IPv4}{-keep}$/ ) {
        if ( exists $opt{i} ) {
            return ( $opt{i} eq 'B' ) ? "$2.$3" : "$2.$3.$4";
        }
        else { return $fdqn_or_ip; }
    }
    else {

        # 最好是用 $RE{net}{domain}{-nospace} 检查，
        # 但（在写本书时）这会强制使用 RFC 1035 规范，而该规范已经被 RFC 1101 更新。
        # 所以像以数字开头的域名（比如 3com.com）会碰到问题。

        # 全部改为小写，以保持一致
        $fdqn_or_ip = lc $fdqn_or_ip;

        # 然后返回第一个点号之后的部分
        $fdqn_or_ip =~ /^([^.]+\.(.*))/;
        return $1;
    }
}
```

这段代码中最有趣的地方是使用了`Regex::Common`解决了很多麻烦。比如用来判断子例程的输入是否为IP地址的匹配，就是借助了这个模块的威力。使用`Regex::Common`意味着我们不必花大力气来编写正则表达式。而`{-keep}`使得正则表达式不但能匹配IP地址，还能匹配地址集合（请参考模块说明文档）：

- `$1`包含整个匹配的内容
- `$2`包含地址的第一部分
- `$3`包含地址的第二部分
- `$4`包含地址的第三部分
- `$5`包含地址的第四部分

我们最初是在第8章中介绍的`Regex::Common`，不过我觉得它非常有用，所以这里再次介绍。

以上就是分析`last`命令的输出并构造数据结构的代码。要完成这个程序的剩余功能，我们还需要遍历所有用户，发现那些登录的域超过指定数量的用户（也就是键的数量过多的用户）。对于这些用户，我们输出如下内容：

```
foreach my $user ( sort keys %userinfo ) {
    if ( scalar keys %{ $userinfo{$user} } > $maxdomains ) {
        print "\n\n$user has logged in from:\n";
        print join( "\n", sort keys %{ $userinfo{$user} } );
    }
}
print "\n";
```

现在你已经看到了完整的代码，可能会感兴趣它实际运行起来如何。下面节选的输出展现了我们是如何发现一个被窃取了密码的用户：

```
username has logged in from:
38.254.131
bu.edu
ccs.neu.ed
dac.neu.ed
hials.no
ipt.a
tnt1.bos1
tnt1.bost
tnt1.dia
tnt2.bos
tnt3.bos
tnt4.bos
toronto4.di
```

其中有些信息对于一个在波士顿的用户来说还比较常见。然而*toronto4.di*就比较可疑，*hials.no*则是一个挪威的地址！

这个程序还能进一步改进以识别时间信息或者关联其他日志文件（比如*tcpwrappers*日志）。不过这里只是要让你看到，模式检测本身已经很有用了。

危险的网络，或者说“Perl挽救了局面”

下面的真实故事展示了Perl是如何在关键时刻得到应用的。在某个周六的晚上我和往常一样登录到网络，开始阅读邮件。但让我惊讶的是，我发现我们的邮件和Web服务器都处于无法工作的状态。收发邮件和访问网页都极为缓慢，不时掉线。邮件队列开始变得非常拥堵。

我立刻查看服务器状态。发现服务器响应还算不错，CPU负载比较高，但是没有达到极限。邮件进程运行的数量非常多，这显示了问题所在。从邮件日志来看，多运行的那些邮件进程是因为很多的事务没有完成。那些启动后处理外部连接请求的进程挂起了，所以导致很高的系统负载。高负载又导致新的连接无法建立。我开始用*netstat*命令分析这个奇怪的网络状况，看看当前都有哪些网络连接。

而netstat输出的最后一列显示有很多外部主机与该服务器进行连接。不过问题在于这些连接的状态。它们不是像下面这样：

tcp	0	0	mailhub.3322	mail.mel.aone.ne.smtp	ESTABLISHED
tcp	0	0	mailhub.3320	edunet.edunet.dk.smtp	CLOSE_WAIT
tcp	0	0	mailhub.1723	kraken.mvnet.wne.smtp	ESTABLISHED
tcp	0	0	mailhub.1709	plover.net.bridg.smtp	CLOSE_WAIT

而是下面的样子：

tcp	0	0	mailhub.3322	mail.mel.aone.ne.smtp	SYN_RCVD
tcp	0	0	mailhub.3320	edunet.edunet.dk.smtp	SYN_RCVD
tcp	0	0	mailhub.1723	kraken.mvnet.wne.smtp	SYN_RCVD
tcp	0	0	mailhub.1709	plover.net.bridg.smtp	CLOSE_WAIT

起初，这看起来像是典型的拒绝服务攻击，名为SYN洪水攻击，或者称为SYN-ACK攻击。要了解这类攻击，我们必须岔开话题讨论一下TCP/IP的工作机制。

每一个TCP/IP连接都是通过参与者之间的握手开始的。这个礼仪使得会话的发起者和接收者都能表达他们愿意加入会话。第一步是由网络会话的发起者发送SYN（意思是SYNchronize，同步）数据包开始。如果接收者愿意开始会话，那么他就会发回SYN-ACK作为对请求的响应，同时在等待连接表中记录这个将要开始的会话。然后发起者再针对SYN-ACK发回ACK数据包，确认自己收到了SYN-ACK。这样，接收者在收到ACK之后可以从自己的等待连接表中删除相应的会话条目，然后开始交谈。

以上的描述就是理想的情况。发生SYN洪水攻击时，恶意客户端会给某台机器发送海量的SYN数据包，并且常常使用假冒的地址。毫无觉察的服务器会开始给所有假冒地址发送SYN-ACK数据包，并且在等待连接表中记录这个会话请求。而这些无效的记录会一直留在那里，直到操作系统的定时清理机制启动。如果发送的攻击数据包足够多，这个等待请求表就会被塞满，这样没有任何连接请求会得到响应。这就导致了我看到的情况，也就是那样的netstat输出。

这里的问题在于netstat输出中有很多主机。一般来说在这种攻击中你会看到为数不多的几个主机，除非攻击者对网络中的主机非常了解，或者攻击者使用了很多的“肉鸡”来进行分布式的拒绝服务攻击。另外，这些主机看上去也都运行正常，并不像成为了“肉鸡”。进一步的测试使得更多的疑问浮出水面，那就是我有时可以ping通（或者traceroute到）netstat输出中的某台机器，有时又不可以。这时候我就迫切需要一个工具进行更广泛的测试，以便了解整个的网络连通状况。这就是Perl大显身手的时候了。

因为我的程序是用来解燃眉之急的，所以我决定写一个快速脚本，尽可能针对问题核心，所以并不担心对外部程序的依赖。现在让我给你看看那个初始版本，然后我们会逐渐把它改进得更为强壮。

任务现在简化成一个问题：我如何尝试连接那些企图连接我的机器呢？为了获取那些企图连接我的服务器的机器，我使用了Brian Mitchell开发的*clog*程序（参考<http://coast.cs.purdue.edu/pub/tools/unix/logutils/clog/>）。*clog*使用了Unix的*libpcap*库（来自Lawrence Berkeley National Laboratory's Network Research Group），从而获得为TCP连接请求（也就是SYN数据包）嗅探网络的能力。著名的网络监控程序*tcpdump*（参考<http://www.tcpdump.org>）也使用了同样的库。这个库能在大多数Unix衍生系统上工作，它的Windows移植则可以在<http://www.winpcap.org>找到。

*clog*能用下面的格式汇报SYN数据包：

```
Mar 02 11:21|192.168.1.51|1074|192.168.1.104|113
Mar 02 11:21|192.168.1.51|1094|192.168.1.104|23
```

上面的输出显示了两个来自192.168.1.51的连接请求，接收者的地址是192.168.1.104。第一个请求尝试连接端口113（*ident*服务），而另一个请求则是尝试连接端口23（*telnet*服务）。

有了*clog*，我就能了解哪些主机正在向我发起连接，现在的问题是如何知道我是否可以连通他们。这个任务应该可以使用一个叫做*fping*的程序来完成，它的作者是Roland J. Schemers III，现在的维护者是Thomas Dzubin。*fping*可以在<http://www.fping.com>下载，它是一个能在Unix衍生系统上运行得更快更好的*ping*程序的替代品。有了这些外部程序，我就写下了这样的Perl程序：

```
use Readonly;

# clog 程序的路径
Readonly my $clogex => '/tmp/clog';

# fping 程序的路径及默认参数
Readonly my $fpingex => '/arch/unix/bin/fping -r1';

Readonly my $localnet => '192.168.1';    # local network
my %cache;

open my $CLOG, '-|', "$clogex" or die "Unable to run clog:$!\n";
while (<$CLOG>) {
    my ( $date, $orighost, $origport, $desthost, $destport ) = split(/\|/);
    next if ( $orighost =~ /^$localnet\b/ );
    next if ( exists $cache{$orighost} );
    print "`$fpingex $orighost`";
    $cache{$orighost} = 1;
}

# 基本上不会运行到此处，因为上面的循环是无限循环，
# 但作为良好习惯，还是在此处写上关闭句柄的操作。
close $CLOG;
```

这个程序先运用*clog*命令，然后从它获得源源不断的输出。因为我们怀疑的不是内部网络连接操作，所以程序会跳过那些原始地址与内网地址一致的连接请求。

我们在代码中引入了一些缓存机制，用来记录曾经检查过的主机，这是为了避免对每台外部主机发出一个以上的*ping*包。另外，*fping*程序被调用的时候也使用了*-r1*标志，这样就关闭了*fping*默认的三次重试模式。

这个程序必须以提升权限(elevated privilege)来运行，因为*clog*和*fping*都需要提升的权限才能访问主机的网络接口。在我的机器上，这个程序输出了如下信息：

```
199.174.175.99 is unreachable
128.148.157.143 is unreachable
204.241.60.5 is alive
199.2.26.116 is unreachable
199.172.62.5 is unreachable
130.111.39.100 is alive
207.70.7.25 is unreachable
198.214.63.11 is alive
129.186.1.10 is alive
```

这显然是很滑稽的事情。怎么解释一半的站点不可以访问，而另外一半可以呢？在我们解答这个问题之前，先来看看我们如何才能改进这个程序。首先我们会避免对外部程序的依赖。如果能从Perl中嗅探网络并且能发送*ping*包，那就会给我们很大的自由。所以现在我们就开始替换外部程序。

Russell Mosemann开发的Net::Ping模块（目前维护者是Steve Peters）和Perl一起发布，能帮我们检查网络连通情况。Net::Ping能让我们发送三种不同的*ping*包：ICMP、TCP以及UDP，并且能检查响应数据包。最典型的ICMP（Internet Control Message Protocol，互联网控制消息协议）响应数据包，也就是大家熟知的*ping*包，往往来自命令行的*ping*程序。使用Net::Ping发送ICMP虽然流行，但这样做有一个重大缺陷，就是需要以提升权限才能运行，类似之前的*clog/fping*代码。

注意：如果你不喜欢受到“以提升权限运行”的限制，我建议你使用Net::Ping::External模块，它的作者是Alexandr Ciornii和Colin McMillen。

Net::Ping::External其实是通过调用操作系统的ping命令（并分析输出）来工作的。因为系统的ping命令往往已经通过某种方式配置过了（比如通过setuid root来设置可执行属性），所以普通用户也可以调用。如果你希望在Windows上实现这个机制，可以使用Toby Ovod-Everett的Win32::PingICMP模块，它能模拟标准的ping命令，通过Win32::API来调用ICMP.DLL。

这个例子中我还是用Net::Ping，因为反正我们也需要以提升权限运行来嗅探网络。不过切换到其他方法也应该是很容易的。

Net::Ping数据包除了通过ICMP发送以外还可以选择TCP（Transmission Control Protocol，传输控制协议）和UDP（User Datagram Protocol，用户数据报协议）。这两者都会把数据包发送到远程主机的*echo*服务端口。使用这两个选项可以降低对权限的要求，不过不如ICMP稳定可靠。这是因为ICMP已经集成在了TCP/IP栈中，而每台主机都可以决定是否要运行*echo*服务。所以ICMP比起其他两种方式更容易收到响应，除非它被禁用了。

Net::Ping使用标准的面向对象语法，所以使用时的第一步是创建一个ping对象实例：

```
use Net::Ping;
my $p = Net::Ping->new('icmp');
```

使用这个对象：

```
if ( $p->ping($host) ) {
    print "ping succeeded.\n";
}
else {
    print "ping failed\n";
}
```

现在让我们看看脚本中真正复杂的部分，也就是网络嗅探（network sniffing）。之前我们使用了*clog*程序来完成这个任务，但这个程序只有Unix版本，所以在其他平台上可能无法工作。如果我们打算让这个功能在Unix以外的平台上工作，就必须选择其他的方法。

通过Perl来实现这个功能的第一步是安装libpcap（对于Windows来说是安装winpcap）库。我建议你同时安装tcpdump。这个程序能直接调用libpcap的功能，而不必事先编写任何Perl代码。

在安装了libpcap之后，Net::Pcap模块（原作者是Peter Lister，后来Tim Potter全部重写，目前维护者是Sébastien Aperghis-Tramoni）的安装也就不难了。这个模块能带给我们完整的libpcap的功能。让我们先看看简单的Net::Pcap的范例，然后再用它来开发类似*clog*的SYN数据包查找程序。

我们的范例代码通过请求指定设备（这里是我的笔记本的无线适配器）的数据包捕获描述符开始：

```
use Net::Pcap qw(:functions);

# 也可以用lookupdev和findalldevs来确定正确的设备
my $dev = 'en1';

# 准备从每个数据包中随意捕获 1500 个字节，
# （即所有传输中的数据，不仅仅是发送给我们的），
# 不限定数据包超时时间，并将错误消息绑定到变量 $err
my $err;
my $pcap = open_live( $dev, 1500, 1, 1, \$err )
```



```
or die "Unable to open_live device $dev: $err\n";
```

注意：如果你需要让脚本更加智能（可以自动发现可监听设备），那么请使用Max Maischein的Sniffer::HTTP模块包中的Net::Pcap::FindDevice模块。在我见过的模块中，这个模块最擅长处理这种需求。

现在我们可以让Net::Pcap开始进行数据包捕获：

```
# 开始捕获数据包，直到人为中断
my $ret = loop( $pcap, ?1, \&printpacketlength, '' );
warn 'Unable to perform capture:' . geterr($pcap) . "\n"
    if ( $ret == ?1 );

Net::Pcap::close($pcap);
```

这就开始了数据包捕获（这里的-1参数意味着直到用户中断才结束，当然也可以设置需要捕获的数据包数量）。每次捕获到数据包之后，我们都会用printpacketlength()中的回调代码进行处理。如果我们不能捕获，那么就打印错误消息并且（礼貌地）尝试关闭与数据包捕获描述符关联的设备。

类似printpacketlength()这样的回调子例程可以从loop()中获取如下数据：

- 用户ID字符串，这个字符串不是必需的，不过如果设置它，可以让多个打开的数据包捕获会话的回调过程彼此区分开
- 一个哈希引用，存储了数据包头的信息（比如时间戳等）
- 一个完整的数据包拷贝

根据这里的第三项数据，我们可以计算出数据包的长度：

```
sub printpacketlength {
    my ( $user_data, $header, $packet ) = @_;
    print length($packet), "\n";
}
```

如果我们运行这里的代码，它就能汇报每个数据包的长度。这是最简单的Net::Pcap使用案例。

好的，现在让我们开始解决SYN数据包捕获问题。libpcap允许你捕获所有的数据包，也允许你设置捕获条件来进行筛选。它的筛选机制非常高效，所以建议尽可能多地使用它，从而避免在Perl代码中进行筛选。在这个例子中，我们只关心SYN数据包。

那么如何识别SYN数据包呢？要做到这点，就必须先了解TCP数据包的格式。图11-1展示了TCP数据包及其包头的结构（该资料参考了RFC 793）。

我们关心的SYN数据包其实就是包头设置中仅有SYN标志（也就是在图11-1中突出显示的）的那些数据包。为了让libpcap只捕获这类数据包，我们需要告诉它需要关注哪个字节。图中的每个刻度线都是一个比特位，所以我们需要计算字节数。图11-2显示了同样的数据包结构，并且标注了字节号。

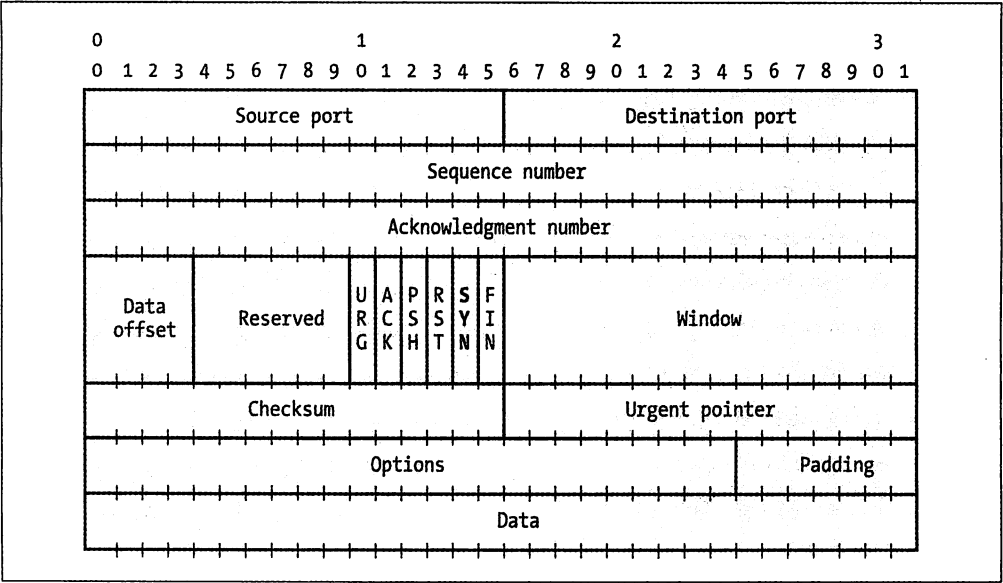


图11-1：TCP数据包结构图

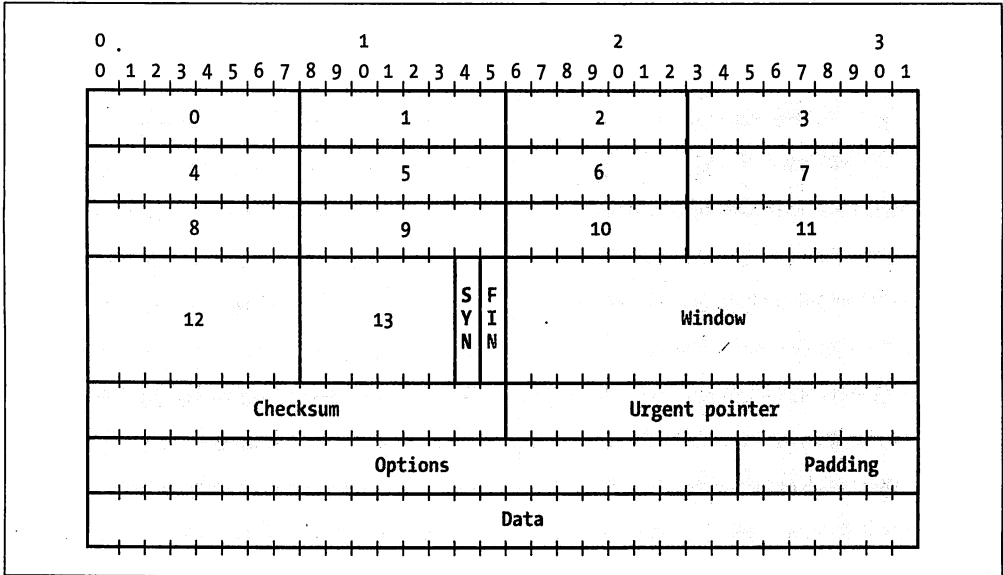


图11-2：找出TCP数据包中正确的字节

所以我们需要检查的是第13个字节是否被置为二进制的00000010，也就是2。这里我们使用了`tcp[13] = 2`作为过滤器。如果我们希望检查至少有SYN标志设置的数据包，那么可以使用`tcp[13] & 2 != 0`这样的过滤器。

只需要在`loop()`调用之前加上几行代码就可以在`Net::Pcap`程序中使用这些信息。把过滤字符串编译成过滤程序，然后对数据包捕获描述符进行设置：

```
my $filter_string = 'tcp[13] = 2';

# 编译并设置我们的“过滤程序”
Net::Pcap::compile( $pcap, \my $filter, $filter_string, 1, 0 )
    and die "unable to compile $filter_string\n";

Net::Pcap::setfilter( $pcap, $filter ) and die "unable to set filter\n";
```

如果我们运行修改过的代码，就能发现数据包长度只是那些只有SYN标志设置的TCP数据包的。

这段代码会捕获SYN数据包并且打印包长度，不过这并不是我们真正需要的。我们需要的程序是能监视来自外网的SYN数据包，然后尝试自动ping发包的远端主机。我们手头已经有了所有的组件，唯一的问题是如何得知SYN数据包的来源地址。

如同我们在第5章中解决DNS问题那样，这里也可以从数据包中直接“拿”出来源地址。当然这往往需要我们阅读RFC规范并且使用`unpack()`模板来完成解析。幸运的是，Tim Potter已经帮我们完成了任务，他的`NetPacket`（包括`NetPacket::Ethernet`、`NetPacket::IP`、`NetPacket::TCP`、`NetPacket::ICMP`等等）系列模块（目前由Yanick Champoux维护）提供了两个方法：`strip()`和`decode()`。

`strip()`能从数据包中移除包头，返回包体。不过要注意的是，常见的以太网上的TCP/IP数据包其实是一个嵌入在以太网数据包中的IP数据包中的TCP数据包。所以，如果`$pkt`中含有一个TCP/IP数据包，那么`NetPacket::Ethernet::strip($pkt)`能够返回IP数据包（去除了以太网层的包头）。如果你需要的是从`$pkt`中解析出TCP层的数据，那么需要做的是使用`NetPacket::IP::strip(NetPacket::Ethernet::strip($packet))`，这样才能除去IP层和以太网层的包头。

而`decode()`则把这个数据包处理过程更推进了一步，它能把数据包的各个组件分解出来，以对象的形式返回。下面的代码行能返回带有表11-2所示的那些字段的对象实例：

```
my $pobj = NetPacket::TCP->decode(
    NetPacket::IP::strip(NetPacket::Ethernet::strip($packet)))
```

表11-2：来自NetPacket::TCP的decode()方法返回的对象的字段

字段名	描述
src_port	源TCP端口
dest_port	目标TCP端口
seqnum	TCP序列号
acknum	TCP响应号
hlen	数据头长度
reserved	6位的TCP数据头“保留”空间
flags	URG、ACK、PSH、RST、SYN和FIN标志位
winsize	TCP滑动窗口大小
cksum	TCP校验和
urg	TCP紧急指针
options	任何TCP选项（二进制格式）
data	这个数据包的真正载荷，也就是数据实体

这些字段应该和你在图11-2中看到的一致。要获取数据包的目标TCP端口，我们只需要这样做：

```
my $dport = NetPacket::TCP->decode(
    NetPacket::IP::strip(
        NetPacket::Ethernet::strip($packet)))->{dest_port};
```

现在让我们把所有的组件都整合起来，并且通过两个小技巧来简化编程任务。Tim Potter在Net::Pcap基础上开发了一个名为Net::PcapUtils的包装模块，简化了必要的初始化和循环代码。它能有效缩短我们的代码。下面就用我们学到的所有技巧来完成这个任务：

```
use Net::PcapUtils;
use NetPacket::Ethernet;
use NetPacket::IP;
use Net::Ping;
use Readonly;

Readonly my $dev => 'en0';
Readonly my $localnet => '192.168.1';

# 过滤字符串，寻找并非来源于本地网络的 SYN 数据包
Readonly my $filter_string => "tcp[13] = 2 and src net not $localnet";

my %cache;

$| = 1; # 禁用 STDIO 缓存
```

```

# 构造稍后使用的 ping 对象
my $p = Net::Ping->new('icmp');

# 准备启程
my $ret = Net::PcapUtils::loop(
    \&grab_ip_and_ping,
    FILTER => $filter_string,
    DEV    => $dev
);
die "Unable to perform capture: $ret\n" if $ret;

# 找出数据包的来源 IP 地址, 并尝试 ping 它 (仅一次)
sub grab_ip_and_ping {
    my ( $arg, $hdr, $pkt ) = @_;

    # 取得来源 IP 地址
    my $src_ip
        = NetPacket::IP->decode( NetPacket::Ethernet::strip($pkt) )->{src_ip};

    print "$src_ip is "
        . ( ( $p->ping($src_ip) ) ? 'alive' : 'unreachable' ) . "\n"
        unless $cache{$src_ip}++;
}

```

如果你喜欢用包装模块来缩短代码, 那么还可以用另一个模块来进一步简化代码。Paul Miller的Net::Pcap::Easy内置的例程可以简化我们每天重复编写的Net::Pcap代码。下面的版本是用它重写代码的结果, 后面有简单的解释:

```

use Net::Pcap::Easy;
use Net::Ping;
use Readonly;

Readonly my $dev      => 'en1';
Readonly my $localnet => "192.168.1";

# 过滤字符串, 寻找并非来源于本地网络的 SYN 数据包
Readonly my $filter_string => "tcp[13] = 2 and src net not $localnet";

my %cache;

$| = 1;    # 禁用 STDIO 缓存

# 构造稍后使用的 ping 对象
my $p = Net::Ping->new('icmp');

# 设置 Net::Pcap 相关的配置, 并准备好回调函数
my $npe = Net::Pcap::Easy->new(
    dev          => $dev,
    filter        => $filter_string,
    packets_per_loop => 10,
    bytes_to_capture => 1500,
    timeout_in_ms  => 1,
    promiscuous    => 1,

```

```

tcp_callback => sub {
    my ( $npe, $ether, $ip, $tcp ) = @_;

    my $src_ip = $ip->{src_ip};

    print "$src_ip is "
        . ( ( $p->ping($src_ip) ) ? 'alive' : 'unreachable' ) . "\n"
        unless $cache{$src_ip}++;
}
);

while (1) { $npe->loop(); }

```

只有最后的代码有改动，所以我们只解释那些被改动的代码。这里最神奇的是`new()`方法，我们一次性设置了捕获参数，包括过滤字符串和回调函数。`Net::Pcap::Easy`让我们能一次定义多种数据包（TCP、UDP、ICMP等等）的回调函数。这里我们只设置了TCP数据包的过滤器，所以只定义了TCP的回调函数。使用`Net::Pcap::Easy`的回调有个好处，就是它会自动进行`NetPacket::*`对象的解析，不必让我们面对原始数据包。所以代码里面的`strip()`和`decode()`就可以免去了。这样我们可以直接处理需要的对象属性，比如`src_ip`。在我们调用`loop()`方法开始循环的时候，在`new()`方法中设置的`packets_per_loop`选项决定了捕获数据包的个数。一旦捕获了足够多的数据包，模块就会把它们交给回调函数，然后退出循环。为了与之前的版本兼容，我们可以反复调用`loop()`，直到用户中断程序。

现在我们已经实现了完全用Perl（虽然有些模块是依赖于C语言代码）来诊断服务器问题，让我来告诉你故事的结尾。

到了周日的早晨，技术支持中心（不是我们部门）发现了一个路由器配置问题。起因是一个学生在宿舍里的机器上安装了Linux，而且他还配置了网络路由服务器。于是这台机器开始向整个大学广播，说它自己是互联网的默认路由器。而我们部门的路由器居然立刻相信了这台机器，并且把自己的路由表修改了，增加了另一个所谓的路由器。对于外界送来的数据包，这个路由器把它们均分给两台机器，“这个数据包给真的路由器，下一个给学生的机器，再下一个给真的路由器，再下一个给学生的机器……”这样导致了所谓的对称路由均分。这个困扰了我们一天的问题在路由表清理之后立刻得到了解决。不过我是不会告诉你那个学生（还有那个配置路由器的员工）后来的遭遇的。

在这一节里，你看到了`Net::Pcap`、`Net::PcapUtils`/`Net::Pcap::Easy`以及`NetPacket::*`系列模块的实际应用。请多多利用这些模块提供的便利性，开发更多的网络监控程序，保护你的网络远离危险。

防范危险行为

守望者的最后一个特质是防范那些不好的事情。其实也就是反复提醒大家：“不要把刚烤好的面包放在窗口冷却。”

这一章最后的案例既可以影响一台机器，又可以改善整个计算环境。另外还有一个显著的不同点，那就是我们不会只满足于继续使用别人的代码，还要构造自己的模块。

这里我的目标是防患于未然，避免那些太差的密码。自古以来任何完善的安全机制都无法面对差的密码。如果oog用户的密码就是“oog”，那么很难保证避免被入侵^[注5]。现存的复杂密码破解工具（比如Solar Designer写的*John the Ripper*以及Alec Muffett的Crack）使得这个问题更加严重。

避免这种问题的方法就是从根源开始避免差的密码。你应该帮助用户选择那些难以破解的密码。这里有两个互补的任务：建议使用合适的密码和避免使用差的密码。

建议更好的密码

选择好的密码之前先要知道什么是好密码。选择差的密码往往是和心理问题、社会问题以及环境方面的因素相关的。其中有种典型的情况是大脑空白。如果有人对你说“赶快找个你能记住的、别人又猜不到的东西”，这显然会让人有压力。

为了避免让人感觉到压力，更好的方法是为他们预先生成建议的密码。有不少Perl模块致力于生成更加安全的密码，其中一些生成的是随机密码，而另外一些则能生成接近随机的密码，不过又可以在某种语言里读出来（这样便于记忆）。随机的密码从理论上说更加安全，不过多年来安全社区一直在讨论到底密码是真正随机（以至于用户要把它写在便利贴上）更好，还是不那么随机但是便于记忆更好。

在这个领域，所有的Perl模块都很容易使用。你可能需要提供一些参数来描述需要的密码类型，或者说明你的语言希望怎样的可读性，不过这样应该就足够了。让我们先看看范例。先是Data::SimplePassword模块，它能打印出随机的10个字符的密码：

```
use Data::SimplePassword;

my $dsp = Data::SimplePassword->new();

# 10 个字符长度的随机密码，也可以用 chars() 方法指定要选用的字符
print $dsp->make_password(10), "\n";
```

如果我需要生成随机密码，我倾向于使用Jörg Walter的Crypt::GeneratePassword模块，

注5： 不过稍后他可能把密码改成了00g。

因为它能生成可读的密码，并且相比之下更加安全，也符合NIST规范（FIPS-181）。另外它还支持对生成密码的过滤，以避免生成不雅的词，因为随机生成的密码可能带有某些让人不舒服的字符序列。使用的时候，我们只要调用`word()`函数就能生成可读的密码，使用`chars()`函数就能生成随机密码。这两个函数都要求两个必要的参数：最短和最长的密码长度。比如下面的代码：

```
use Crypt::GeneratePassword;

for (1..5) {
    print Crypt::GeneratePassword::word( 8, 8 ), "\n";
}
```

可能生成下面的输出：

```
ecloorfi
neleappw
xchanedo
noutoone
nopenule
```

拒绝差密码

给用户建议好的密码只是一个开始，不过万一用户忽视你的建议，还必须要有办法拒绝差密码。对于Unix主机来说，有个办法就是使用Alec Muffett的*CrackLib*。通过开发*Crack*，Muffett给系统管理社区提供了一个绝佳的工具，因为我们可以利用它的基于C语言的破解机制。

这个库的接口只有一个函数：`FascistCheck()`。这个函数带有两个参数：一个需要检查的字符串，以及*CrackLib*安装时导入的字典文件的全路径。这个函数返回两种信息，要么返回NULL，意味着这个密码是安全的；要么返回一段文本，说明这个密码不安全的原因（比如“is a dictionary word”）。这个功能对设置或者修改密码的Perl程序来说非常有用，所以让我们尝试把它包装成模块。这个小练习需要和C代码打交道，不过这个过程并不复杂。

注意：出于扩大知识面的需要，我需要指出CPAN上其实已经有一个可用的模块：Dan Sully写的*Crypt::Cracklib*。在写本书的第一版的时候我还不知道此模块的存在，所以在版本更新的时候我就加入了这段信息，不过这一节还是保留了下来，因为我想这个实践对希望创建自己模块的人是有益的。Sully的模块确实能工作，不过我还是建议那些不想用它的人自己动手创建。

小提示：在写本书的时候，这个模块的测试还不能匹配新版*CrackLib*的响应信息，所以你可能需要强行安装。

我们第一步要做的是编译*CrackLib*软件包，它来自<http://sourceforge.net/projects/cracklib>。这个软件包中的描述非常简单，我列出了三个要点：

- 对于这个软件包需要的字典文件来说，你能收集到越多越好。这个字典文件中的词汇表有两个好的来源，一个是<ftp://ftp.ox.ac.uk/pub/wordlists>，另一个是Openwall项目销售的词汇表CD，在<http://www.openwall.com/wordlists>。词典的构造过程需要不少的临时磁盘空间（出于*utils/mkdict*中排序进程的需要），所以请提前做好资源。
- 请确保构造*CrackLib*的时候使用和构造Perl一样的开发工具。比如说，如果你编译Perl的时候使用的是*gcc*，那么在编译*CrackLib*的时候也请使用*gcc*。这个要求对于其他C语言模块的编译也是一样的。
- 这一节的范例代码使用了*CrackLib*的2.8.12版本。请确保你使用的版本足够新，哪怕是需要删除操作系统中预先安装的版本也在所不惜。

一旦我们编译好了*libcrack.a*这个C语言库程序（或者等效共享库）之后，我们就需要设置从Perl中调用库里的*FascistCheck()*函数的方法。这个方法被称为XS。^[注6]

最简单的XS用法是通过*h2xs*程序来创建模块原型：

```
$ h2xs -b 5.6.0 -A -n Cracklib
Writing Cracklib/ppport.h
Writing Cracklib/lib/Cracklib.pm
Writing Cracklib/Cracklib.xs
Writing Cracklib/Makefile.PL
Writing Cracklib/README
Writing Cracklib/t/Cracklib.t
Writing Cracklib/Changes
Writing Cracklib/MANIFEST
```

表11-3描述了这个命令创建的文件。

表11-3：h2xs -b 5.6.0 -A -n Cracklib命令创建的文件

文件名	描述
<i>Cracklib/ppport.h</i>	跨版本可移植的头文件
<i>Cracklib/lib/Cracklib.pm</i>	Perl“树桩”（stub）程序和文档
<i>Cracklib/Cracklib.xs</i>	C代码黏合器
<i>Cracklib/Makefile.PL</i>	生成Makefile所用的Perl代码
<i>Cracklib/t/Cracklib.t</i>	“树桩”测试代码

注6：在本书第一版中我还推荐使用SWIG作为接口方法，不过目前来说这种用法已经逐渐在Perl社区衰败了，最起码CPAN上的模块看来如此。

表11-3：h2xs -b 5.6.0 -A -n Cracklib命令创建的文件（续）

文件名	描述
Cracklib/Changes	版本说明文档
Cracklib/MANIFEST	模块包含的文件列表

我们只需要修改这些文件中的几个就能集成需要的功能。让我们从最难的部分开始，也就是C代码黏合。下面展示如何在*crack.h*这个头文件中定义*CrackLib*函数：

```
const char *FascistCheck(const char *pw, const char *dictpath);
```

警告： 为了节约你的时间，这儿有一个提醒：XS工具对空格相当敏感，所以如果你是自己在家里做练习，那么请注意在拷贝*Cracklib/Cracklib.xs*代码的时候小心对待空格。

在我们的*Cracklib/Cracklib.xs*黏合文件中，我们重复这个定义：

```
#include <crack.h>

PROTOTYPES: ENABLE

const char *
FascistCheck(pw,dictpath)
    char *pw
    char *dictpath
```

这里的PROTOTYPES指令用来给黏合文件中的函数创建Perl原型。这并不是我们代码的需要，引入它只是为了避免构造过程中的警告。

在函数定义之后，我们定义它如何被调用以及返回的内容：

```
CODE:

    RETVAL = FascistCheck((const char*)pw, (const char*)dictpath);

OUTPUT:
    RETVAL
```

RETVAL是这个文件中真正的黏合点。它代表了C代码和Perl解释器之间的转运站。这里我们告诉Perl它需要接收从FascistCheck()C库函数返回的一个字符串并且让这个字符串作为Cracklib::FascistCheck()这个Perl函数的返回值（即OUTPUT）。

然后我们还需要从文件中删除由h2xs工具给这个文件加入的#include "ppport.h"行。因为我们这里做的事情和它无关，也是说不需要Devel::PPPort模块的支持。其实如果你用Perl解析器来运行这个头文件，它会告诉你哪些是真正需要的：

```

$ perl ppport.h
Scanning ./Cracklib.xs ...
=== Analyzing ./Cracklib.xs ===
No need to include 'ppport.h'
Suggested changes:
--- ./Cracklib.xs      2009-01-03 22:08:28.000000000 ?0500
+++ ./Cracklib.xs.patched  2009-01-03 22:08:30.000000000 ?0500
@@ ?2,7 +2,6 @@
    #include "perl.h"
    #include "XSUB.h"

-#include "ppport.h"

#include <crack.h>

```

当我们把这个文件从*Cracklib/Cracklib.xs*里面删除的时候，我们也应该把它从*Cracklib/MANIFEST*中一并删除。

以上就是我们需要接触的C代码。

其他的文件都只需要修改其中的几行就可以了。为了确保Perl能找到libcrack库以及它的头文件*crack.h*，我们需要修改*Cracklib/Makefile.PL*文件中WriteMakefile()调用时的参数。下面就是其中需要增加和修改的行：

```

LIBS          => [''], # e.g., '-lm'
DEFINE        => '', # e.g., '-DHAVE_SOMETHING'
MYEXTLIB      => '/opt/local/lib/libcrack${LIB_EXT}',
INC           => '-I. -I/opt/local/include',

```

以上就是我们需要做的必要修改，模块应该已经准备好了。^[注7]现在我们只要输入：

```

$ perl Makefile.PL
$ make
$ make install

```

就可以这样使用模块了：

```

use Cracklib;
use Term::Prompt;
use Readonly;

Readonly my $dictpath => '/opt/local/share/cracklib/pw_dict';

my $pw = prompt( 'p', 'Please enter password:', '', '' );
print "\n";

my $result = Cracklib::FascistCheck( $pw, $dictpath );
if ( defined $result ) {

```

注7：我在自己的苹果电脑上需要为Mac OS X的*macports*追加-lintl这个LIBS定义，不过这并不是常见环境的需要，所以我没有把它放在范例代码中。

```

    print "That is not a valid password because $result.\n";
}
else {
    print "That password is peachy, thanks!\n";
}

```

先别着急使用这个模块，让我们把它变成一个更加专业的作品，然后再正式安装它。

首先，我们要修改h2xs创建的测试脚本来检查模块是否正常工作。这里我们需要改用功能更加齐全的测试模块，也就是用Test::More。Test::More帮我们生成支持测试装置（test harness）的输出。使用这个模块之后，我们只需要做两件事：

- 指定计划测试的个数，也就是修改tests => 1那一行。
- 用is()函数来调用我们的函数，参数应该是我们期望得到的结果。

下面就是修改之后的Cracklib/t/Cracklib.t文件的内容。我已经从其中拿掉了例行的h2xs生成的注释，使得文件更加容易阅读：

```

use Test::More tests => 6;
BEGIN { use_ok 'Cracklib' };

# 我们的 cracklib 字典文件所在路径
#
# 为了让测试文件具备可移植性，
# 我们应该把它改成该模块中的文件，然后指定相对路径
my $dictpath = '/opt/local/share/cracklib/pw_dict';

# 用于测试的字符串及其已知的cracklib响应
my %tests =
(
    'happy'      => 'it is too short',
    'a'          => 'it is WAY too short',
    'asdfasdf'   => 'it does not contain enough DIFFERENT characters',
    'minicomputer' => 'it is based on a dictionary word',
    '1ftm2tgr3fts' => undef;
);

foreach my $pw (sort keys %tests){
    is(Cracklib::FascistCheck($pw,$dictpath), $tests{$pw}, "Passwd = $pw");
}

```

现在我们可以输入make test，这样Makefile就会进行必要的测试，验证模块是否正常工作：

```

PERL_DL_NONLAZY=1 /opt/local/bin/perl "-MExtUtils::
Command::MM" "-e" "test_harness(0, 'blib/lib', 'blib/arch')" t/*.t
t/Cracklib....ok
All tests successful.
Files=1, Tests=6, 0 wallclock secs ( 0.02 cusr + 0.01 csys = 0.03 CPU)

```

测试脚本固然重要，但我们的脚本只有配合说明文档才能真正派上用场。所以同时有必

要改进的还有`Cracklib/Cracklib.pm`和`Cracklib/Changes`文件中的“树桩”信息。另外，仔细编辑`Cracklib/README`文件^[注8]以及用来描述如何安装的`Cracklib/INSTALL`文件，也是很好的想法。其中还可以介绍如何安装相关的组件（比如`CrackLib`）以及哪里有范例代码。所有那些新加入的文件也应该在`Cracklib/MANIFEST`文件中登记，以便模块编译代码正确工作。

最后，还要在你接触的环境中尽可能多地安装这个模块。这样，无论在哪里遇到与这个模块相关的需求，都可以很方便地使用它。随着你的工作环境中差密码越来越罕见，守望者也会越来越高兴。

本章所用模块

模块名	CPAN ID	版本
Getopt::Std (随Perl发布)		1.06
Digest::SHA	MSHELOR	5.47
Net::DNS	OLAF	0.64
FreezeThaw	ILYAZ	0.43
File::Find (随Perl发布)		1.13
File::Find::Rule	RCLAMP	0.30
Regexp::Common	ABIGAIL	2.122
Net::Ping (随Perl发布)	SMPETERS	2.35
Net::Pcap	SAPER	0.16
Net::PcapUtils	TIMPOTTER	0.01
NetPacket	YANICK	0.41
Net::Pcap::Easy	JETTERO	1.32
Data::SimplePassword	RYOCHIN	0.04
Crypt::GeneratePassword	JWALT	0.03
Readonly	ROODE	1.03
Term::Prompt	PERSICOM	1.04

注8：我还有一个小小的心愿是修改`h2xs`生成的`README`文件，不能把它直接发布到CPAN上面去，那样会显得模块的质量低劣（也会给作者丢脸）。而且换个角度说，修改这个小文件也费不了多少精力。那么我们要在其中写点什么呢？

更多参考资料

<http://www.tcpdump.org>是libpcap和tcpdump的主页，winpcap可以在<http://www.winpcap.org>找到。

RFC 793: Transmission Control Protocol，由J. Postel所著（1981年），即TCP说明文档。

《MD5 To Be Considered Harmful Someday》，Dan Kaminsky所著（2004年），可以在http://www.doxpara.com/md5_someday.pdf中找到。

http://www.perlmonks.org/?displaytype=print;node_id=431702是个关于如何编写自己的模块的很有趣的指南（只是有一点过时了）。

“建议更好的密码”一节采用自我原本发表在《;login》杂志专栏中的内容，专栏名为“This Column is Password Protected”。

RFC 1321: The MD5 Message-Digest Algorithm，由R. Rivest所著（1992年），即MD5说明文档。

FIPS 180-2: Secure Hash Standard (SHS) 记录了SHA-1和SHA-2标准（本书写作之时），可以从<http://csrc.nist.gov/publications/fips/fips180-2/fips180-2withchangenotice.pdf>下载。

tripwire曾是用来监测文件系统改动的权威的免费软件。在它商业化之后，它的公司停止了将文件系统改动监测软件单独作为产品销售。其他一些开源软件，如yafic (<http://www.saddi.com/software/yafic/>) 和AIDE(<http://www.cs.tut.fi/~rammer/aide.html>)就出现了，填补了该空白。

SNMP (Simple Network Management Protocol, 简单网络管理协议) 提供了远程监控和配置网络设备及网络主机的方法。一旦掌握了SNMP的基础, 你就可以随时监控 (和配置) 网络中每台设备和主机的运行。

不过老实说, “简单” 网络管理协议并不简单。如果你不熟悉SNMP, 可能还会有些吃力, 不过可以参考附录G的教程来快速入门。

从Perl中使用SNMP

一个从Perl来使用SNMP的方法是调用命令程序。在附录G中我介绍了如何使用Net-SNMP软件包里面的命令程序。这个方法非常简单, 类似于我们在本书中一再展示的其他外部程序调用, 所以我们就不在这方面过多介绍了。

但有个例外情况: 如果你在使用SNMPv1或者SNMPv2c, 可能会考虑把团体名放在命令行中。但如果这个程序放在多用户环境中运行, 任何人都可以从进程列表中看到你的密码。这种危险在附录G中展示的命令行中也同样存在, 而且对于那些定时自动启动的程序来说会更加严重。为了方便演示, 这一章的范例会简单地用主机名和团体名作为命令行参数, 不过你不应该把它用在生产环境中^[注1]。

如果不打算通过外部程序调用来从Perl执行SNMP操作, 那么还可以使用Perl的SNMP模块。可以选择的至少有三个类似的模块: David M. Town写的`Net::SNMP`, Simon Leinen写的`SNMP_Session.pm`, 以及一个有着众多名字的模块 (包括`NetSNMP`、`Perl/SNMP`以及`Net-SNMP`库的SNMP扩展模块第五版), 此模块的原始作者是G. S. Marzot, 目前由Net-SNMP项目维护。我们会把这个模块称为SNMP, 因为它正是以这个

注1: 另外还需要对`snmp.conf`采用特殊手段进行保护, 这在Net-SNMP包的说明文档中都介绍了。

名字载入Perl程序的。所有这三个模块都支持SNMPv1，而Net::SNMP和SNMP还能支持某些SNMPv2c及SNMPv3的特性。表12-1列出了这三个模块从Perl调用Net-SNMP命令行工具的对比。

表 12-1：从Perl调用SNMP的几种方法对比

功能	SNMP_Session	Net::SNMP	SNMP	Net-SNMP 命令行
SNMPv1支持	是	是	是	是
SNMPv2c支持	是	是	是	是
SNMPv3支持	否	是	是	是
OID解析	否	否	是	是
发送第一版的trap	是	是	是	是
接收第一版的trap	是	否	否	是
发送第二版的notification	是	是	是	是
接收第二版的notification	是	否	否	是
发送第三版的notification	否	否	是	是
接收第三版的notification	否	否	否	是
发送inform	否	是	是	是
接收inform	是	否	否	是
完全使用Perl实现	是	v1和v2c是， v3不是	否	是

除了对SNMP的支持程度有差异之外，这三个模块最大的区别在于它们对外部库程序的依赖性。Net::SNMP和SNMP_Session.pm主要是用Perl来实现的^[注2]，而SNMP则依赖于Net-SNMP库程序。这会导致SNMP在构造时有额外的依赖性问题要注意，因为你必须先在自己的平台上构造Net-SNMP库程序。

不过额外的依赖性也给模块带来了额外功能。比如SNMP可以分析MIB（Management Information Base,管理信息库）描述文件，也能打印出原始的SNMP数据包（供调试用），而前两个模块则没有这些功能。另外还有一些模块能补足这个功能差异，不过如果你希望用一个模块来解决所有问题，那么SNMP可能是最好的选择。

注2： 如果你在SNMPv3环境下使用Net::SNMP，它会依赖于几个基于C的模块（比如Crypt::DES、Digest::MD5和Digest::SHA1Digest::SHA1），所以它并非纯Perl模块。

注意：在安装的时候请特别注意尽可能安装Net-SNMP源代码包中*perl*目录下的SNMP模块。CPAN上的版本往往比这个版本老，所以有可能不会和当前Net-SNMP库程序同步。

让我们从一个小的Perl例子开始介绍。如果需要了解某个设备的接口数量，可以查询它的*interfaces.ifNumber*变量。使用Net::SNMP模块非常简单，代码如下：

```
use Net::SNMP;

# 需要提供主机名和团体名作为参数
my ($session,$error) = Net::SNMP->session(Hostname => $ARGV[0],
                                           Community => $ARGV[1]);

die "session error: $error" unless ($session);

# iso.org.dod.internet.mgmt.mib-2.interfaces.ifNumber.0 =
# 1.3.6.1.2.1.2.1.0
my $result = $session->get_request('1.3.6.1.2.1.2.1.0');

die 'request error: '.$session->error unless (defined $result);

$session->close;

print 'Number of interfaces: '.$result->{'1.3.6.1.2.1.2.1.0'}."\n";
```

如果对网络中某台主机做这样的查询，可能的输出是Number of interfaces: 2；对于带有以太网接口、回环接口和PPP接口的笔记本来说输出是Number of interfaces: 3；对于一个小型路由器来说输出是Number of interfaces: 7。

这里要特别注意的是程序使用了OID（对象标识符）而非变量名。因为Net::SNMP和SNMP_Session.pm这两个模块都只关心SNMP协议，并不会帮你把可读的变量名翻译成OID，因为那个工作需要额外分析MIB的描述信息。为了实现这个目的，你可以使用其他模块，比如SNMP::MIB::Compiler或者Mike Mitchell写的SNMP_util.pm（这个模块必须和SNMP_Session.pm一起使用）^[注3]。

如果你希望直接使用文本标识符（而不是数字OID），那么你唯一的选择就是使用SNMP模块，因为它既不需要自己编程来完成转换，也不依赖于外部模块。现在让我们使用这个模块导出某台机器的ARP（Address Resolution Protocol，地址解析协议）表：

```
use SNMP;

# 需要提供主机名和团体名作为参数
my $session = new SNMP::Session(DestHost    => $ARGV[0],
                                Community   => $ARGV[1],
```

注3： 注意，不要把SNMP_util.pm和Wayne Marquette编写的名字相似的SNMP::Util模块混在一起。Marquette的模块是用来支持SNMP模块而创建的，和SNMP_util.pm的作用大不相同。

```

        Version      => '1',
        UseSprintValue => 1);

die "session creation error: $SNMP::Session::ErrorStr" unless
    (defined $session);

# 为 getnext() 命令设置数据结构
my $vars = new SNMP::VarList(['ipNetToMediaNetAddress'],
                              ['ipNetToMediaPhysAddress']);

# 取出第一行
my ($ip,$mac) = $session->getnext($vars);
die $session->{ErrorStr} if ($session->{ErrorStr});

# 以及后续所有行
while (!$session->{ErrorStr} and
        $vars->[0]->tag eq 'ipNetToMediaNetAddress'){
    print "$ip -> $mac\n";
    ($ip, $mac) = $session->getnext($vars);
};

```

下面就是这段程序的样本输出：

```

192.168.1.70 -> 8:0:20:21:40:51
192.168.1.74 -> 8:0:20:76:7c:85
192.168.1.98 -> 0:c0:95:e0:5c:1c

```

这段代码类似于之前的Net::SNMP版本。我们现在开始逐段分析差别：

```

use SNMP;

my $session = new SNMP::Session(DestHost      => $ARGV[0],
                                Community      => $ARGV[1],
                                Version        => '1',
                                UseSprintValue => 1);

```

在载入SNMP模块之后，我们创建了一个会话对象，这类似于我们在Net::SNMP范例中所做的。这里附加的Version => 1参数设置了协议的版本号（默认是第三版），而参数UseSprintValue => 1则能让SNMP模块对输出内容进行优化。如果不设置它，那么输出中的以太网（MAC）地址就会采用原始格式。

接下来的一行创建了后面getnext()方法将要用到的对象：

```

my $vars = new SNMP::VarList(['ipNetToMediaNetAddress'],
                              ['ipNetToMediaPhysAddress']);

```

SNMP在命令中支持interfaces.ifNumber.0这样的字符串，不过对于getnext()请求来说必须先创建一个名为VarBind的特殊对象。在RFC 1157中谈到变量绑定时说：“变量绑定，也可以称为VarBind，指的是一对变量名和变量值的关系。VarBindList则是一组变量名和对应值的列表。”如果你听到这里开始联想Perl的哈希键—值对，那么你就对

了。虽然在SNMP模块中没有使用哈希和哈希的列表来实现这两个概念，但核心思想是类似的。所以接下来的代码用VarList()方法生成一个对象，其中包含一个列表，列表中又含有两个VarBind。每个变量绑定都可以理解成一个匿名数组的引用，其中含有一个obj元素。

VarBind实际上是通过Perl匿名数组来实现的，数组中有四个元素，分别是obj、iid、val和type。这样的设计是因为这更加类似于SNMP编码系统的数据结构。对于常规任务来说，最重要的是obj和iid。第一个元素，也就是obj，代表了你在查询的对象。obj可以用几种方式来表达，这里我们使用的是叶标识符（leaf identifier）格式，也就是只列出路径的叶子部分。ipNetToMediaNetAddress就是下面树的叶子（路径太长了，必须折行显示）：

```
.iso.org.dod.internet.mgmt.mib-2.ip.ipNetToMediaTable.  
ipNetToMediaEntry.ipNetToMediaNetAddress
```

VarBind的第二个元素是iid，也就是实例标识符。在之前的所有范例中，我们总是使用0，比如system.sysDescr.0。这是因为我们处理的对象总是只有一个实例。我们马上就能看到iid不是0的例子。比如在多接口的以太网交换机上，就需要用它来识别具体的某个网络接口。

对于VarBind来说，obj和iid通常只是在对get()操作时才有必要说明。对于其他操作SNMP会自动补足这些值。如果你使用getnext()调用，那么连iid也是可以省略的，因为这个方法会自动返回下一个实例。这正是为什么之前的代码可以只设定VarBind的第一个元素（也就是obj）就能创建两个用来获取VarList的VarBind。

从我们的角度来看，你可以把VarBind理解成与SNMP查询打交道的数据窗口。比如，之前的代码调用了getnext()方法来发送GetNextRequest请求，类似于我们在附录G中的范例。我们获取的结果是用来发出getnext()调用的索引。SNMP通过VarBind来存放返回的iid，这样我们就不必自己维护最新的值。下次的getnext()调用会自动返回下两个VarBind，其中有我们需要的值，这样我们就取到了需要的信息。

然后我们把创建的VarList对象作为参数传给getnext()方法：

```
# 取出第一行  
my ($ip,$mac) = $session->getnext($vars);  
die $session->{ErrorStr} if ($session->{ErrorStr});
```

getnext()返回从查询中获取的值，并同时修改VarList数据结构。现在我们只需要不断调用getnext()，直到我们取完所有的数据：

```
while (!$session->{ErrorStr} and  
      ($vars->[0]->tag eq 'ipNetToMediaNetAddress')){
```

```

    print "$ip -> $mac\n";
    ($ip,$mac) = $session->getnext($vars);
};

```

作为最后一个与SNMP模块相关的范例，我们会使用一个安全领域的场景。使用SNMP命令行工具来完成任务会非常复杂，最起码也会让人感到麻烦。

想象一下如下场景：你得到一个任务，在交换以太网网络跟踪一个可疑的用户，唯一的信息就是那个用户机器的以太网地址。那个以太网卡地址并非预先登记的（如同在第5章中展示的），而你也难以对交换网络进行嗅探。在这种情况下，最常见的解决方法是逐个询问所有的以太网交换机，看看它们是否认识这个地址（也就是说这个地址是否在交换机的动态CAM表中）。手动完成这个任务是一个大麻烦，因为需要逐个连到网络设备并且运行多个命令。

为了让范例更加实在，以便提供确定的MIB变量名，我们假定网络中主要是思科Catalyst 6500和4500交换机。这里介绍的方法对其他供应商的产品应该也能奏效，另外我们也会在产品相关的部分给出特别提示。现在让我们开始分析问题。

如同以往那样，我们需要从正确的MIB模块文件中搜索信息。通过咨询思科的技术支持，我们得知数据分散在五个对象中：

- 首先是vmMembershipTable，它的位置在这里（路径太长，必须折行显示）：

```

enterprises.cisco.ciscoMgmt.ciscoVlanMembershipMIB.
ciscoVlanMembershipMIBObjects.vmMembership

```

它是在CISCO-VLAN-MEMBERSHIP-MIB中描述的^[注4]。

- dot1dTpFdbTable（透明端口转发表），在RFC 1493 *BRIDGE-MIB*中描述，位置是dot1dBridge.dot1dTp。
- dot1dBasePortTable，在同一个RFC中描述，位置是dot1dBridge.dot1dBase。
- ifXTable，在RFC 1573 *IF-MIB*（接口）中描述。
- vlanTrunkPortTable，位置是：

```

enterprises.cisco.ciscoMgmt.ciscoVtpMIB.vtpMIBObjects.vlanTrunkPorts

```

描述于CISCO-VTP-MIB。

为什么分散在五个表中？确实这里每一张表都只能解答部分问题，没有专门设计好的一张表可查。第一张表给我们提供了所有VLAN（Virtual Local Area Network，虚拟局域

注4： 在本书第一版，我们用了CISCO-STACK-MIB中的vlanTable。这对于老的思科设备来说还能工作，但对于新设备来说，vmMembershipTable 是唯一的信息来源。

网)的信息,也就是所谓的交换导致的“网络分段”^[注5]。在这里思科选择为交换机上的每个VLAN设计一组表,所以我们需要逐个对VLAN进行查询(稍后深入介绍)。

第二个表给我们提供了以太网地址和最后看到此地址的交换机桥接端口对照表。不幸的是,桥接端口号只是交换机内部的概念,并不直接对应到交换机的物理端口。因为我们需要的恰恰是交换机连接的那张网卡的物理端口名,所以还得进一步查表。

没有(想象中的)桥接端口和物理端口名的直接映射表,但是有一个dot1dBasePortTable,它能提供桥接端口和接口编号的映射。一旦有了端口号,我们就可以查询ifXTable,从而获得端口名。

最后,我们使用vlanTrunkPortTable来判断某个接口是否是主干接口(也就是用来和其他主干网络设备互联用的)。我们可以忽略从所有的主干端口上观察到的以太网地址,因为这个端口会汇报从另一个主干网络学习来的地址,而这个信息对我们分解到具体端口并无益处。

图12-1展示了一个执行我们的预期任务所需的四层解引用结构。

下面的代码把这五张表放在一起并转储我们感兴趣的信息:

```
use SNMP;

my ($switchname, $community, $macaddr) = @ARGV;

# 以下是我们需要的所有 MIB并附注了原因
$ENV{'MIBS'}=join(':', ('CISCO-VLAN-MEMBERSHIP-MIB', # VLAN 清单和状态
                        'BRIDGE-MIB',                # MAC 地址到端口的映射表
                        'CISCO-VTP-MIB',              # 主干端口状态
                        ));

# 连接到交换机,取得 VLAN 的列表
$session = new SNMP::Session(DestHost => $switchname,
                              Community => $community,
                              Version   => 1);

die "session creation error: $SNMP::Session::ErrorStr" unless
    (defined $session);

# enterprises.cisco.ciscoMgmt.
# ciscoVlanMembershipMIB.ciscoVlanMembershipMIBObjects.vmMembership.
# vmMembershipTable.vmMembershipEntry
# 位于 CISCO-VLAN-MEMBERSHIP-MIB 内
my $vars = new SNMP::VarList (['vmVlan'], ['vmPortStatus']);
```

注5: 从技术角度说, VLAN 其实是“广播域”,不过大多数人把它理解成网络分区方法,因为这个技术使得 VLAN 中某台主机只能看到同一 VLAN 中其他主机的通信。

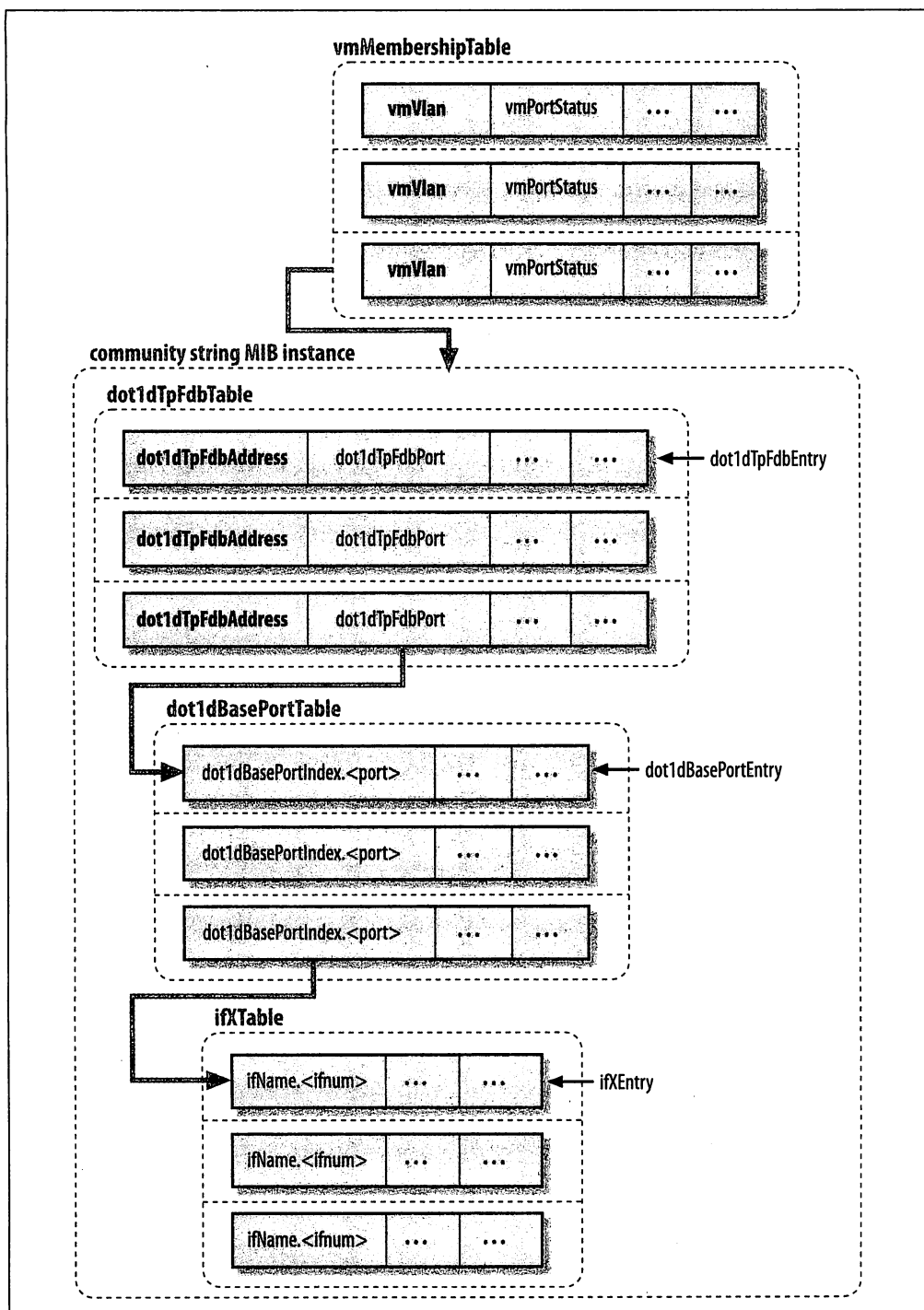


图12-1：用于在思科6500或者4500上查询端口名的SNMP查询序列

```

my ( $vlan, $vlanstatus ) = $session->getnext($vars);
die $session->{ErrorStr} if ($session->{ErrorStr});

my %vlans;
while (!$session->{ErrorStr} and $vars->[0]->tag eq 'vmVlan'){
    $vlans{$vlan}++ if $vlanstatus == 2; # make sure the vlan is active (2)
    ( $vlan, $vlanstatus ) = $session->getnext($vars);
};

undef $session,$vars;

# 确认 MAC 地址正确无误
my $findaddr = message_mac($macaddr);

# 对每一个 VLAN, 查看是否有一个桥接端口看到过特定的MAC地址,
# 如果有, 则查询与此端口关联的接口号, 进而取得此接口号对应的接口名称
foreach my $vlan (sort keys %vlans) {

    # 用于缓存查询结果
    # (我们仅对单个 VLAN 保留缓存数据)
    my (%ifnum, %portname);

    # 注意, 设置会话时, 我们用到了当前 VLAN 的名称作为团体名
    my $session = new SNMP::Session(DestHost => $switchname,
                                     Community => $community.'@'.$vlan,
                                     UseSprintValue => 1,
                                     Version      => 1);

    die "session creation error: $SNMP::Session::ErrorStr"
        unless (defined $session);

    # 查看 MAC 地址是否位于桥接转发表
    # 注意: $macaddr 必须是 XX.XX.XX.XX.XX.XX 这样的格式
    #
    # 通过透明转发端口表查询, 它位于
    # dot1dBridge.dot1dTp.dot1dTpFdbTable.dot1dTpFdbEntry
    # 在 RFC 1493 BRIDGE-MIB 中描述
    my $portnum = $session->get(['dot1dTpFdbPort',$findaddr]);

    # 要是当前VLAN内没有, 则转投下一个VLAN
    next if $session->{ErrorStr} =~ /noSuchName/;

    # 将转发表中的端口号转换成接口编号
    #
    # 通过 dot1dBridge.dot1dBase.dot1dBasePortTable.dot1dBasePortEntry 查询
    # 在 RFC 1493 BRIDGE-MIB 中描述

    my $ifnum =
        (exists $ifnum{$portnum}) ? $ifnum{$portnum} :
        ($ifnum{$portnum} =
         $session->get(['dot1dBasePortIfIndex',$portnum]));

    # 如果该接口是主干端口则跳过
    #
    # 通过 ciscoVtpMIB.vtpMIBObjects.vlanTrunkPorts.vlanTrunkPortTable 查询。

```

```

# vlanTrunkPortEntry 在 CISCO-VTP-MIB 中定义
next if
    $session->get(['vlanTrunkPortDynamicStatus',$ifnum]) eq 'trunking';

# 转换端口号为端口名 (即module/port)
#
# 通过 ifMIB.ifMIBObjects.ifXTable.ifXEntry 查询, 在 RFC 1573 IF-MIB 中描述
my $portname =
    (exists $portname{$ifnum}) ? $portname{$ifnum} :
    ($portname{$ifnum}=$session->get(['ifName',$ifnum]));

print "$macaddr on VLAN $vlan at $portname\n";
}

# 接受以 XX:XX:XX:XX:XX:XX、XX-XX-XX-XX-XX-XX或 XXXXXXXXXXXXXXXX
# (X 表示十六进制数) 作为格式的MAC地址, 然后以十进制返回, 以便后续查询
sub message_mac {
    my $macaddr = shift;

    # 如果传入的是一串没有标点符号的字符, 则人为追加冒号分隔
    $macaddr =~ s/(.)(?=.)/$1:/g if (length($macaddr) == 12);

    # 不管是以冒号还是破折号分隔的地址, 最终都切分后转换成十进制地址
    return join('.', map (hex,split('/:','-']/uc $macaddr)))
}

```

如果你已经读了附录G, 那么这里的代码应该并不陌生。下面我们看看有哪些是需要解释的:

```

$ENV{'MIBS'}=join(':', ('CISCO-VLAN-MEMBERSHIP-MIB', # VLAN 清单和状态
                        'BRIDGE-MIB',                # MAC 地址到端口的映射表
                        'CISCO-VTP-MIB',              # 主干端口状态
                    ));

```

这段代码设置了MIBS环境变量, 这是为了让Net-SNMP能分析那些额外的MIB模块, 获取对象定义信息。这些文件都应该在Net-SNMP的默认搜索路径中。如果你不想把所有文件都放在一个位置, 那么可以设置MIBFILES环境变量来使文件位置更加自由。

注意: 对于Net-SNMP的MIBS环境变量有些常见的误解。我在这方面的理解也不是很深入, 所以我必须尽早把概念理清, 以免你渐渐混淆。MIBS变量中包含的是SNMP MIB模块名, 而不是SNMP MIB模块的文件名。模块名一般是在文件的第一行 (不含注释) 声明的。比如在思科发布的文件*CISCO-VLAN-MEMBERSHIP-MIB-VISMI.my*中, 第一行是这么写的:

```
CISCO-VLAN-MEMBERSHIP-MIB DEFINITIONS ::= BEGIN
```

所以这里我们只能用CISCO-VLAN-MEMBERSHIP-MIB来设置MIB环境变量。

你可能已经注意到, 虽然程序需要引用IF-MIB和BRIDGE-MIB中的对象, 但我们并没有引

入这两个模块，因为它们属于Net-SNMP默认载入的MIB模块。这个默认载入的列表是在Net-SNMP编译时的配置过程中决定的，因为IF-MIB和BRIDGE-MIB都是建议的选项。不过你还是可以显式地用MIBS列出它们，这能让程序更加清晰。但这个决定纯属个人喜好。

继续往下看代码，这里还有一段奇怪的语句：

```
foreach my $vlan (sort keys %vlans) {  
    my $session = new SNMP::Session(DestHost => $switchname,  
                                     Community => $community.'@'.$vlan,  
                                     UseSprintValue => 1,  
                                     Version      => 1);
```

这里没有直接使用用户输入的团体名，而是在后面拼接@ VLAN-NUMBER这样的字符串。在思科的术语中，这被称为“团体字符串索引”。在VLAN和桥接的处理上，思科决定为每个设备维护多组MIB“实例”，也就是每个设备有多组MIB拷贝，每个VLAN一组。我们的代码必须对每组VLAN实例进行查询。下面就是两个查询：

```
my $portnum = $session->get(['dot1dTpFdbPort',$findaddr]);  
  
# 要是当前VLAN内没有，则转投下一个VLAN  
next if $session->{ErrorStr} =~ /noSuchName/;
```

另外还有：

```
my $ifnum =  
    (exists $ifnum{$portnum}) ? $ifnum{$portnum} :  
    ($ifnum{$portnum} =  
     $session->get(['dot1dBasePortIfIndex',$portnum]));
```

第一段代码中需要特别注意的是我们用来查询dot1dTpFdbTable的变量\$macaddr，其中的以太网地址格式是非常规的。这是因为dot1dTpFdbAddress表中的MAC地址采用的是点分十进制格式：NNN.NNN.NNN.NNN.NNN.NNN。为此，我们引入了message_mac()子例程来处理这样的工作，此子例程输入的是常规的MAC地址格式，输出的是可以用来查询的格式。

在第二段代码中，我们实现了简单的缓存机制。在每次get()操作之前，我们先查询哈希表%ifnum，看看其中是否有查询的记录。如果确认没有，我们才会真的进行查询。这个技巧在SNMP编程时值得注意，因为它能有效降低查询的数量和频率，减轻网络和设备的负担。如果我们随意发起查询，那么网络设备的性能可能会有显著下降。

下面就是整段程序的输出：

```
00:60:b0:b7:1e:ed on VLAN 116 at "Gi4/2"
```

不难看出这个程序值得改进。除了对输出进行美化以外，还有必要存储运行结果。在每次运行的时候，这个程序应该可以汇报最新的变动：新发现的地址、端口的变动等等。另外有一个小提议：大多数的交换机都有一种学习机制，因此它们会在一段时间之后“忘记”那些不常用的地址（对于典型的思科设备来说，这个时间是三分钟）。这意味着你的程序的运行频率（最起码）应该比这个“忘记”频率高。

发送和接收SNMP Trap、Notification和Inform

使用Perl与SNMP trap、notification、inform打交道非常简单，所以这一节会非常精简。快速回顾附录G的内容，trap、notification和inform使得SNMP代理（在v1和v2c中的称呼）或者SNMP实体（在v3中的称呼）能够把重要信息直接发送到管理终端，不必先等待查询。也就是说，可以异步发送重要信息，比如说“嗨，我着火了！”或者更加常见的信息如“某台路由器出问题了”。协议中需要一种特殊的方式来传递这些信息，因为这种信息要么非常紧急无法等待查询后再发出，要么不适合定时查询，想象一下这样的对话“你着火了么？你着火了么？现在怎么样了？还没着火么？”这种信息在SNMPv1叫做traps，在v2和v3叫做notification^[注6]。

inform可以说就是notification的增强版。对多数SNMP notification来说，发送设备只是把信息发送到管理终端，然后“交互”就结束了。问题是这种信息往往是用UDP协议传输的，就设计目的来说^[注7]，UDP并没有义务保证接收者确实收到此信息。SNMPv2c为此引入了inform来加入响应机制，有时也被称为“确认的notification”。如果管理终端收到了“合法”信息（请参考RFC中关于合法的定义），它就会向发送者响应以便确认。不过你应该想到了，这个响应信息也是用UDP传输的，所以仍然不能确保送达。RFC中没有定义在这种情况下发送方应该怎么做，不过这总比没有这个机制要好。

现在让我们快速演示如何使用Perl来发送和接收trap、notification、inform。我们先看看如何发送，因为这是最常见的操作。

如同在附录G中提到的那样，trap消息的格式从SNMPv1到v2有了较大的变动（应该是这个变动导致trap到了v2被改名为“notification”）。不过幸运的是，发送的过程几乎完全相同。下面的代码能够使用SNMP模块发送v1的trap信息：

```
my $s = new SNMP::TrapSession(..., Version => 1);
$s->trap(enterprise => '.1.3.6.1.4.1.2021',    # Net-SNMP MIB 扩展
        agent      => '192.168.0.1',
```

注6： 我曾经见到有人用“trap notification”来统称两种情况。

注7： 除非传输使用的是TCP，虽然在RFC 3430中规定这是可行的（RFC 3417中也被列为可行的传输方式），但实际上这非常罕见。

```

generic    => 2,                # 链接断开
specific   => 0,
uptime     => 1097679379,        # 可以改用当前时间
[['ifIndex', 1, 1],             # 哪个接口
 ['sysLocation', 0, 'dieselcafe']]); # 在哪个位置

```

SNMP::TrapSession()和SNMP::Session()有相同的参数（DestHost、Community等等，用省略号代替）。Version在这里用于指定发送的是SNMPv1的trap。而发送SNMPv2的notification的代码则更加容易读懂：

```

my $s = new SNMP::TrapSession(..., Version => '2c');
$s->trap(oid    => 'linkDown',
        uptime => 1097679379,        # 可以改用当前时间
        [['ifIndex', 1, 1],         # 哪个接口
         ['ifAdminStatus', 1, 1],    # 可进行管理
         ['ifOperStatus', 1, 2]]];   # 但不能操作

```

发送SNMPv3的inform的代码几乎和notification的代码相同（SNMPv2的inform目前还没有实现）。下面的代码突出了差别所在：

```

sub callback {...};
my $s = new SNMP::TrapSession(..., Version => '3');
$s->inform(oid    => 'linkDown',
          uptime => 1097679379,        # 可以改用当前时间
          [['ifIndex', 1, 1],         # 哪个接口
           ['ifAdminStatus', 1, 1],    # 可进行管理
           ['ifOperStatus', 1, 2]],    # 但不能操作
          [\&callback, $s]);

```

从2c版本的trap切换到第3版的inform的过程还是比较容易理解的，不过回调函数看起来有些特别。这里设计回调代码是为了接收响应信息。在响应送达或者超时的时候，一个叫做“callback”（随你心意，也可以叫做“message”或“got_it”）的子例程会被调用，其中的参数表明到底是响应送达还是超时。可以参考SNMP模块的文档了解参数的详细信息。另外关于这段代码还有两个需要解释的地方：

- 你不必总是使用回调代码，因为最后一个参数是可选的。你其实可以在发送需要响应的请求之后立刻结束程序，不必等待响应，这是你的自由。这里我能想到使用inform而不是notification的唯一原因应该是企业的政策有时候会规定我们只能使用informs，而不是notification。
- 这里提到回调函数是为了介绍一种使用SNMP模块的新方法。SNMP模块的大多数方法（比如get()、getnext()、set()等等）都可以接受回调函数引用作为最后的可选参数。一旦使用这个引用，这些方法就会在异步模式下运行。也就是说在后台运行，不必等待响应才返回。这里我们不会真的使用这个模式，只是介绍它的原理。一般情况下，get()调用会导致程序挂起并等待，直到响应送达（或者超时）才返

回。但是在异步模式下，程序发送了请求之后就立刻返回，可以继续执行其他语句。一旦响应送达，之前指定的回调代码就开始运行，并且从参数中获取之前请求的结果。这种模式能通过后台运行来避免整个程序挂起，所以往往能提高程序效率。最典型的应用场合就是网络管理GUI编程，在你执行后台SNMP查询的时候，用户仍然可以对窗口进行操作。

现在你已经看到了如何发送trap、notification和inform，很自然地会感兴趣如何接收这些信息。相比之下，接收的任务就不那么常见了，因为大多数机构都会购买一个大型的网络监控软件包来接收这些信息并且发出警报。不过这些软件要么昂贵，要么臃肿，对于简单任务往往处理得过于复杂，所以让我们看看如何自己编写接收程序。

最简单平常的方法是启用*snmptrapd*，并且分析它的输出（这个程序在Net-SNMP软件包中）。这个方法虽然并不吸引人，但是目前为止对SNMPv3来说是唯一可行的Perl方式。不过稍后我们会用一个有创意的方式来调用它，所以请耐心等待。

如果你需要使用纯Perl程序来处理SNMPv1的trap和v2c的notification，可以看看这一章目前还没介绍的模块*SNMP_Session.pm*，作者是Simon Leinen。这个模块广为人知的原因主要是它与MRTG（Multi Router Traffic Grapher，多路由器流量图示器）这个网络监控程序的关联，不过其实这个模块本身已经非常有用了。目前为止*SNMP_Session.pm*还不能从CPAN下载，所以请参考这章末尾的模块信息来了解如何获取它。

下面是从*SNMP_Session.pm*模块的说明文档中摘录的SNMPv1的trap监听程序：

```
use SNMP_Session;
use BER;
my $trap_session = SNMPv1_Session->open_trap_session()
    or die 'cannot open trap session';
my ($trap, $sender_addr, $sender_port) = $trap_session->receive_trap()
    or die 'cannot receive trap';
my ($community, $enterprise, $agent,
    $generic, $specific, $sysUptime, $bindings) =
    $trap_session->decode_trap_request($trap)
    or die 'cannot decode trap received';
...
# this is how we would decode the bindings (e.g., if dealing
# with v2c notification)
my ($binding, $oid, $value);
while ($bindings ne '') {
    ($binding,$bindings) = decode_sequence($bindings);
    ($oid, $value) = decode_by_template($binding, "%O%@" );
    print BER::pretty_oid($oid), ' => ',pretty_print ($value),"\n";
}
```

我们首先开启一个会话，然后等待接收会话的数据。一旦收到数据，就把它解码成名为VarBind的数据字段（OID-值对）。类似于之前例子中的数据访问机制，我们逐个遍历一系列的数据对。对于SNMPv2c的notification来说，接收方法也非常相似（只需要把SNMPv1_Session替换成SNMPv2c_Session即可），主要的区别在于关键数据的编码方法。对于v1来说，大多数的数据在decode_trap_request()调用之后已经返回了。虽然还可以继续使用绑定变量遍历剩下的数据，但是绝大多数的关键信息早就已经获得了。对于v2的notification来说，关键的信息完全在绑定变量中，需要通过遍历才能获取，所以我们在遍历之前做了一个空的解码。

在我们进入下个主题之前，我还要介绍一个令人兴奋的接收trap和notification的技巧。从Net-SNMP软件包的5.2版开始，我们可以在snmptrapd中编译内嵌的Perl解析器。如果在snmptrapd的配置文件中加入perl ...指令，那么启动的守护进程就可以在收到trap、notification和inform的时候调用你指定的代码（比如调用某个子例程）。这个解决方案应该能理想地解决跨领域的问题，因为现在你不必处理复杂的网络监听、数据接收、解码、后台守候方式运行等等问题。你只要专心用Perl写这些事件的处理程序就可以了。

其他SNMP编程接口

现在我们已经展示了所有常规的使用Perl进行SNMP编程的方法。一旦你掌握了SNMP的精髓，这些技术就非常好用。不过对于之前展示的跨表查询的例子，还是常常让人想到该如何简化代码。在这一节，我们会介绍一些其他的编程途径，它们能让编程任务得到简化。不过你还是应该经常去<http://search.cpan.org>搜索“snmp”这个关键词，看看最近又有什么新模块出现。

有些模块能帮你省去记忆常用的SNMP变量名或者OID的工作。它们提供了方法来返回最常见的查询结果。比如Jonathan Stowe写的Net::SNMP::Interfaces，以及James Macfarlane写的Net::SNMP::HostInfo。它们分别在Net::SNMP的基础上提供了以下这些方法调用：

```
$interface->ifInOctets()  
$interface->ifOperStatus()  
$interface->ifOutErrors()
```

另外还有：

```
$hostinfo->ipForwarding()  
$hostinfo->ipRouteTable()  
$hostinfo->icmpInEchos()
```

John D. Shearer写的SNMP::BridgeQuery模块也能通过Net::SNMP来访问桥接设备（比如网络交换机）的某些特殊表。有了它，你只要调用一个函数就能获得设备的桥接转发表

或者地址转换表。

与此一脉相承的更复杂的模块还有SNMP::Info，这起初是Max Baker为了netdisco项目而设计的一个模块。现在SNMP::Info成了供应商（或者设备）相关子模块的开发框架。

```
SNMP::Info::Layer1::Allied
SNMP::Info::Layer2::Aironet
SNMP::Info::Layer2::Bay
SNMP::Info::Layer2::HP
SNMP::Info::Layer3::Foundry
SNMP::Info::Layer3::C6500
```

使用SNMP::Info之后，无需了解供应商相关的SNMP变量名就可以进行许多常规查询，比如查询接口的全双工模式。这意味着如下简单的代码就能奏效：

```
use SNMP::Info;

my $c = SNMP::Info->new(AutoSpecify => 1,
                        DestHost    => $ARGV[0],
                        Community   => $ARGV[1],
                        Version     => '2c');

my $duplextable = $c->i_duplex();

print "Duplex setting for interface $ARGV[2]: " .
      $duplextable->{$ARGV[2]} . "\n";
```

这段代码接受主机名、团体名以及接口号作为参数，并返回相应接口的全双工模式。代码清晰易读，这是因为：

- 我们不必写大量的与厂商/设备相关的代码来罗列需要查询的所有设备，因为每个设备都有不同的SNMP变量名或者OID。
- 我们甚至不必检查供应商或者设备的型号，只要在创建对象时设置AutoSpecify => 1就行了，模块会自己完成相关检查。
- 要了解全双工模式的列表，我们也不必写大量的getnext()调用来完成表的遍历，只要调用一个函数就行了。

对于大多数编程难题来说，SNMP::Info是值得考虑的解决方案。因为它真的可以让简单网络管理协议变得简单起来。

本章所用模块

模块名	CPAN ID/URL	版本
Net::SNMP	DTOWN	5.01
SNMP	http://www.net-snmp.org	5.2.1
SNMP_Session.pm	http://www.switch.ch/misc/leinen/snmp/perl/	1.07
SNMP::MIB::Compiler	FTASSIN	0.05
SNMP_util.pm	http://www.switch.ch/misc/leinen/snmp/perl/	1.04
SNMP::Util	WMARQ	1.8
Net::SNMP::Interfaces	JSTOWE	1.1
Net::SNMP::HostInfo	JMACFARLA	0.04
SNMP::BridgeQuery	JSHEARER	0.58
SNMP::Info	MAXB	0.90

更多参考资料

有超过70份的RFC文档在标题中包含了“SNMP”（并且有超过100份文档是在其他地方提及SNMP）。下面是本章及附录G中引用到的RFC文档：

- *RFC 1157: A Simple Network Management Protocol (SNMP)*, 由J. Case、M. Fedor、M. Schoffstall和J. Davin所著（1990年）
- *RFC 1213: Management Information Base for Network Management of TCP/IP-based Internets: MIB-II*, 由K. McCloghrie和M. Rose所著（1991年）
- *RFC 1493: Definitions of Managed Objects for Bridges*, 由E. Decker、P. Langille、A. Rijssinghani和K. McCloghrie所著（1993年）
- *RFC 1573: Evolution of the Interfaces Group of MIB-II*, 由K. McCloghrie和F. Kastenholz 所著（1994年）
- *RFC 1905: Protocol Operations for Version 2 of the Simple Network Management Protocol (SNMPv2)*, 由J. Case、K. McCloghrie、M. Rose和S. Waldbusser所著（1996年）
- *RFC 1907: Management Information Base for Version 2 of the Simple Network Management Protocol (SNMPv2)*, 由J. Case、K. McCloghrie、M. Rose和S. Waldbusser所著（1996年）

- *RFC 2011: SNMPv2 Management Information Base for the Internet Protocol using SMIPv2*, 由K. McCloghrie 所著 (1996年)
- *RFC 2012: SNMPv2 Management Information Base for the Transmission Control Protocol using SMIPv2*, 由K. McCloghrie所著 (1996年)
- *RFC 2013: SNMPv2 Management Information Base for the User Datagram Protocol using SMIPv2*, 由K. McCloghrie所著 (1996年)
- *RFC 2274: User-based Security Model (USM) for Version 3 of the Simple Network Management Protocol (SNMPv3)*, 由U. Blumenthal和B. Wijnen所著 (1998年)
- *RFC 2275: View-based Access Control Model (VACM) for the Simple Network Management Protocol (SNMP)*, 由 B. Wijnen、R. Presuhn 和 K. McCloghrie 所著 (1998 年)
- *RFC 2578: Structure of Management Information Version 2 (SMIPv2)*, 由 K. McCloghrie、D. Perkins 和 J. Schoenwaelder 所著 (1999 年)

还有其他很多关于 SNMP 的很好的资源。

<http://www.simpleweb.org>是一个关于网络管理的很棒的集合站点，其中有很大一块内容就是关于SNMP的。

<http://net-snmp.sourceforge.net>是Net-SNMP项目的主页。

<http://www.cisco.com/public/sw-center/netmgmt/cmtk/mibs.shtml>是Cisco的MIB文件的地址。其他的厂商也有类似的站点。

<http://www.snmpinfo.com>是SNMPinfo公司和David Perkins（一位SNMP专家，经常给 *comp.protocols.snmp* 寄信，同时也是《Understanding SNMP MIBs》的作者之一）的主页。

<http://www.ibr.cs.tu-bs.de/ietf/snmpv3/>是关于 SNMPv3的非常好的资源。

<http://www.mrtg.org>和<http://cricket.sourceforge.net>是MRTG和派生项目Cricket（以 Perl 编写）的主页，它们是两个通过SNMP来对设备进行长期监控的好例子。

《Understanding SNMP MIBs》，由David Perkins和Evan McGinnis所著（Prentice Hall 出版），是关于MIB的很好的资源。

<http://www.snmp.org>是SNMP Research公司的主页。该网站的“protocol”部分有很多很好的参考资料，包括 *comp.protocols.snmp* 邮件列表的常见问题列表。

网络映射和监控

相信只要是管理过网络的人，哪怕没有正式的“网络管理员”名号，也会在乎两个最基本的问题：“网络中有哪些节点？”（也就是关于网络映射的问题）以及“这些节点是否在做正确的事情？”（也就是关于网络监控的问题）。你有可能会认为第一个问题相对来说简单一些（因为毕竟只是你自己的网络），但在如今这个年代，网络世界充斥着不到20美元的迷你hub以及各种无线访问，所以解答起来也并非那么容易。另一方面，要确保Web服务器总是在提供HTTP或者HTTPS服务、路由器总是在正确吞吐数据包、数据库服务器一直在线服务，这些确实很重要。但是，可能更为重要的是要能知道什么时候开始Web服务器突然开始提供SMTP服务、数据库服务器开始支持Web服务、路由器突然开始丢包等等。这一章就是解答网络映射和监控问题的。我们的目标是帮你识别并理解那些用来构造解决方案的组件。

网络映射

我们首先尝试解决的是网络映射问题，因为只有解决了这个问题才能开始监控。在计算机的中生代，检查网络映射非常容易。那时候你所需要的全部工具就是铅笔和白纸，再加上几分钟的安静时间。因为网络中没几台机器，而且全部是你自己安装的，安装和配置服务器对普通用户来说太复杂了。不过，如今几乎所有人都可以不费任何力气把笔记本电脑接入网络，并开启各种类型的网络会话。

好了，让我们结束怀旧的情绪，回到冰冷的现实，让我们看看到底哪些是我们感兴趣的网络映射。可供选择的项目虽多，但真正常见的并不多：

- 现存的主机
- 网络设备的配置情况
- 网络拓扑结构

- 网络服务
- 网络主机的物理位置

在这些项目中，最后一个是最常见的问题，也是最复杂的。所以我们可以先看看其他那些比较简单的项目，积累足够经验之后再讨论它。

发现主机

在识别网络主机时，我们可以用两种互补的方式来实现：主动或被动。主动方式意味着要发送某种数据包到网络，而被动方式则只需要监听。让我们看看如何进行这些处理，然后比较它们的优缺点。

最简单的常规主动探测方法就是向某段网址发送ICMP ECHO_REQUEST数据报（也就是“ping”包），然后监听ICMP ECHO_RESPONSE数据报。有几个模块能简化发送ping包的过程：

Net::Ping

这可以说是所有Ping模块的祖父。这些年来，这个模块逐渐扩展成可以从ICMP以外的协议发送类似ping的数据包。因为原作者Russell Mosemann、其他的贡献者以及目前的维护人员（Rob Brown）都在不断改进这个模块，使它能发送数据包至主机TCP的TCP或UDP echo服务，能发送标准ICMP请求，现在它甚至能发送部分的TCP握手数据包。它还能调用我们下一个要介绍的模块。

Net::Ping::External

常规的ping包发送往往会给Perl脚本带来困难。虽然大多数主机更容易响应ICMP数据包（而不是对TCP或UDP echo服务的请求）^[注1]，但是大多数操作系统都要求从事ICMP相关活动的脚本以提升权限运行。Net::Ping::External填补了这样的空白，它帮助那些权限不足的脚本调用操作系统内置的ping可执行程序。那些内置的可执行程序往往已经被赋予了足够权限，也能免于安全检查。Net::Ping::External为这个跨平台的调用实现了兼容，所以你的脚本可以不必担心如何调用可执行程序，也不必考虑不同平台的输入输出差异。

Win32::PingICMP

如果通过调用外部程序来绕过权限问题的思路让你不舒服，其实还有一种让Perl用户更加舒服的方法，不过它只能在Windows平台生效。在Windows平台具备原

注1： 随着时间的推移，到了如今杂草丛生的互联网年代，ICMP开始被越来越多的网络管理员和主机拒绝。比如Windows XP SP2以上的防火墙就会拒绝ICMP，不过如果你使用Net::Ping的syn协议模式来进行探测就能绕过防火墙。

始socket处理能力之前，这种操作系统必须通过*ICMP.dll*来实现ICMP数据包的发送和接收。虽然微软的文档没有声明，但这个DLL能被那些普通权限的程序（比如标准的ping程序）调用，从而完成相应的任务。Toby Ovod-Everett编写的Win32::PingICMP模块就能调用这个DLL。这使得Perl在处理ICMP数据包收发的时候能得到（比ping程序的输出）更精确的时间。不过有个情况要注意，微软已经声明这个DLL会从操作系统中消失，所以如果这种情况真的发生，那么这个模块就不能继续工作。

现在让我们来看看这个模块是如何工作的。下面的代码展示了如何使用ping来对某个网段进行“扫荡”。它使用了Net::Netmask模块来进行网段内地址的计算：

```
use Net::Ping;
use Net::Netmask;

my $ping = Net::Ping->new('icmp');    # 必须使用超级用户的身份来运行该脚本

# 提供需要扫描的网络及掩码
die $Net::Netmask::error
    unless my $netblock = new2 Net::Netmask( $ARGV[0] );

my $blocksize = $netblock->size() - 1;

# 这个循环的运行需要一点时间，因为某些无法通信的地址要等响应超时了才能判断
my @addrs;
for ( my $i = 1; $i <= $blocksize; $i++ ) {
    my $addr = $netblock->nth($i);
    push( @addrs, $addr ) if $ping->ping( $addr, 1 );
}
print "Found\n", join( "\n", @addrs ), "\n" if scalar @addrs;
```

以上的Net::Ping代码非常容易读懂，只不过是一个new()调用，然后是ping()某个地址而已。所以我们主要关注的是范例中的Net::Netmask方法。首先使用new2来创建一个Net::Netmask对象。这里的new2和常规的新之间的主要区别是创建的对象如何处理错误数据：如果收到不能理解的输入new2会返回undef，而new则返回空对象。我觉得这个区别非常细微（我更希望程序在收到错误数据的时候直接抛出异常），目前范例里暂且先用new2。这个对象有几个有用的方法调用，比如size能返回地址的数量，而nth能返回网段内的第N个地址。这使得遍历地址变得很容易，我们可以对每个地址发出ping调用。Net::Netmask还有一个enumerate()方法可以这样调用：

```
for my $address ( $netblock->enumerate ) {...}
```

但这对于巨型网络来说会有危险，因为可能产生太大的项目列表。

另外一个相关的技术是ARP（Address Resolution Protocol，地址解析协议）扫描。ARP是用来帮助主机识别本地网段中的另一台主机的硬件地址的。协议规定主机可以广播这

样的问题：“哪台主机是192.168.0.11？”而持有那个地址的主机可以回答：“我在这里，我的卡号是00:1e:c2:c2:a1:f1。”要进行ARP扫描，就必须对网段中所有的IP地址发送广播，然后等待主机的回答。之所以说这个方法有些危险，原因有二：

- ARP是一种重要的网络基础协议。如果你的程序影响了它的正常工作（无论是类似ARP欺骗攻击那样的有意破坏，还是类似ARP广播风暴那样的无意干扰），会产生非常坏的后果。所以在考虑采用这种方法之前请三思。
- 某些操作系统（比如Windows）在遇到对“人造”的ARP请求的响应（也就是操作系统自己没有请求过的响应）时会非常生气，尤其是针对主机的当前地址。有多么生气呢？如果你从拥有某个IP地址的机器查询自己的IP地址，那么操作系统会关闭这个网络接口，然后大肆报警。所以千万不要使用ARP查询自己的IP地址。

现在我已经警告过了，让我们开始扫描主机吧。不过要想做到跨平台地创建ARP数据包、发送ARP数据包并监听响并不容易（主要是考虑到不同平台的安装和运行时的差异）。下面我们会尝试三种方式，其中起码应该有一种能在你的环境下工作。

这看上去有些滑稽，不过最接近跨平台方法的ARP处理机制确实是使用外部程序。Web上有不少数据包构造方面的工具套件，包括spak、ipsend、rain、arp-sk、hping和nemesis。我们打算介绍最后一个，因为它的作者Mark Grimes和Jeff Nathan花了不少精力来确保nemesis能在BSD、Linux、Solaris、OS X和Windows上运行。其他的工具套件的跨平台性都不及它。

警告： nemesis目前只能支持libnet 1.0.2版。而大多数的Linux发行包安装的都是更新的libnet 1.1版。使nemesis可以在老的libnet版本运行的方法可以参考<http://codeidol.com/security/anti-hacker-tool-kit/TCP-IP-Stack-Tools/NEMESIS-PACKET-WEAVING-101>。

使用nemesis创建ARP数据包非常容易：

```
nemesis arp -v -S 192.168.0.2 -D 192.168.0.1
```

这个命令会返回类似下面的信息：

```
ARP/RARP Packet Injection --- The NEMESIS Project Version 1.4 (Build 26)
```

```
      [MAC] 00:0A:95:F5:92:56 > FF:FF:FF:FF:FF:FF
[Ethernet type] ARP (0x0806)
```

```
      [Protocol addr:IP] 192.168.0.4 > 192.168.0.1
[Hardware addr:MAC] 00:0a:95:f5:92:56 > 00:00:00:00:00:00
      [ARP opcode] Request
[ARP hardware fmt] Ethernet (1)
[ARP proto format] IP (0x0800)
```

```
[ARP protocol len] 6
[ARP hardware len] 4
```

这个程序很容易用Perl脚本来调用，类似于之前演示的`Net::Ping`调用。不过它和`Net::Ping`代码有一个重大差异，那就是我们得自己处理返回的响应。我们之前在第11章中已经看到了如何捕获数据包。在监听ARP响应数据包的时候，我们可以用这个过滤器：

```
arp[7]=2
```

现在让我们看看这个过滤器是如何工作的。其中的=2应该很好理解：RFC 826中定义了ARP响应的操作码（`ares_op$REPLY`）应该是2。而困难的是使用`proto[N]`这样的表示法来确定要查找数据包中的什么地方。不过如果手头有了类似图13-1这样的ARP数据包结构图之后应该不难。

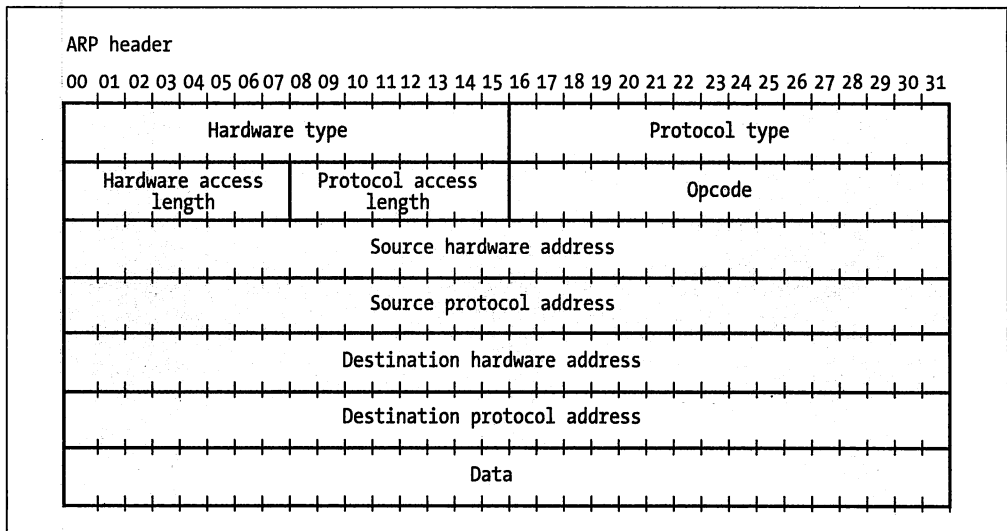


图13-1：ARP包结构图

除了调用外部程序以外，还有两个可选的ARP扫描方法。首先是尝试用我们在第5章中展示 的模块自己动手实现。它能帮你创建ARP请求并且从网络发送出去。`Net::Packet`模块能支持ARP数据包的创建。而`Net::Pcap`模块能读取收到的响应。我们已经展现了这些模块的工作机制，所以我们直接跳到最后一个方法。

Oleg Prokopyev的`Net::Arping`模块提供了最后一种ARP扫描的方法。它和前两种方法的共同之处是同样依赖于Mike Schiffman的`libnet`^[注2]和`libpcap`（原作者是Lawrence Berkeley

注2：之前对`nemesis`依赖的`libnet`库的版本限制（只支持1.0.2版）对于`Net::Arping`模块也适用。

National Laboratory的Van Jacobson、Craig Leres和Steven McCanne，目前由*tcpdump.org*的志愿者维护）。Windows用户可以使用libpcap的特殊升级版，名为WinPcap。

Net::Arping使用的语法接近于之前展示的Net::Ping的：

```
use Net::Arping;

my $arping = Net::Arping->new();
# 如果收到ARP响应，arping()会返回其中的MAC地址
my $return = $arping->arping($ARGV[0]);
print "$ARGV[0] " .
    ($return) ? "($return) is up\n" : "is down\n";
```

现在我们可能已经开始对扫描类型的模块感觉厌烦了，所以现在让我们换个角度，看看被动方式的网络映射分析。这种方式不要求对主机发送扫描信息，往往也需要运行更久才能得到结果。不过这个方式在以下环境下非常适用：

- 在希望避免扫描类型的操作时。嗅探类型的操作往往有些诡异的部件让人厌烦。
- 在不希望增加网络负载的时候。这对于那些已经很慢的或饱和线路特别适用。
- 在不希望对网络造成破坏的时候。之前已经提到了，某些类型的ARP数据包可能带来危害。相比之下被动方式不容易导致这种问题。

我们要展示的第一个被动方式可以称为“静听”。虽然这个名字听上去类似于某种自省的机制，但是对于网络监测也适用。这个方法其实是用第11章介绍的数据包嗅探技术来对适合的东西进行监听。那么哪些是适合监听的东西呢？这取决于你的网络以及你要嗅探网络上的何处，下面列出了常见的项目：

- ARP流量
- DHCP租用/续租请求和服务器响应
- 来往于中心服务器的流量
- SNMP或者其他网络监控请求

这里列出的项目除了最后一项，大部分都比较容易理解，所以我们先弄清楚它再开始写代码。我的意思是如果某种网络监控程序正在扫描网络设备的状况，那么监听它所收发的数据包就很有价值。你可能会觉得很奇怪，如果网络中已经有了监控程序，为什么我们还要写自己的程序呢？记住，做这件事情的主要目的是为映射整个网络和/或找出缺少的那一个，而那些监控程序往往只对一小组设备感兴趣。所以应该充分利用系统中现存的资源来帮助我们映射网络，网络信息的来源越多越好。

现在让我们看看嗅探网络并显示请求和回复的ARP流量。注意，这段代码（和其他的网络嗅探程序一样）需要以提升权限运行才能工作：

```

use Net::PcapUtils;
use NetPacket::Ethernet;
use NetPacket::ARP;

my $filter = 'arp';
my $dev = 'en1'; # device for my wireless card
my %addresses = ();

die 'Unable to perform capture: ' . Net::Pcap::geterr($dev) . "\n"
    if ( Net::PcapUtils::loop(\&CollectPackets,
                               FILTER => $filter,
                               DEV => $dev,
                               NUMPACKETS => 100,
                               )
    );

print join( "\n", keys %addresses ), "\n";

sub CollectPackets {
    my ( $arg, $hdr, $pkt ) = @_ ;

    # 将原来十六进制的协议地址（即 IP 地址）
    # 转换为以点区分的形式（即 X.X.X.X）
    my $ip_addr = join(
        '.',
        unpack(
            'C*',
            pack( 'H*',
                NetPacket::ARP->decode( NetPacket::Ethernet::strip($pkt) )
                ->{'spa'} )
            )
        );

    $addresses{$ip_addr}++;
}

```

这段代码在运行之后会监听100个ARP数据包，并从其中分析出IP地址。它也会打印出直到结束为止发现的所有主机。对所监听的网络数据包类型的修改其实就是对libpcap过滤器的修改（当然还有包解析代码）。比如下面的代码就能发现DHCP响应流量，并且显示发现的DHCP服务器：

```

use Net::PcapUtils;
use NetPacket::Ethernet;
use NetPacket::IP;

my $filter = 'dst port 68'; # DHCP 响应端口
my $dev = 'en1'; # 我的无线网卡设备名称

my %addresses = ();

die 'Unable to perform capture: ' . Net::Pcap::geterr($dev) . "\n"
    if (
        Net::PcapUtils::loop(

```

```

        \&CollectPackets,
        FILTER      => $filter,
        DEV         => $dev,      # 我的无线网卡设备
        NUMPACKETS => 100,
    )
);

print join( "\n", keys %addresses ), "\n";

sub CollectPackets {
    my ( $arg, $hdr, $pkt ) = @_ ;

    # 将十六进制的IP地址转成分形式
    my $ip_addr =
        NetPacket::IP->decode( NetPacket::Ethernet::strip($pkt) )->{'src_ip'};

    $addresses{$ip_addr}++;
}

```

这段代码非常简单。它并不会尝试解释ARP或者DHCP数据包的内容，因为这不是重点。重点是需要知道哪台机器正在发送某种协议的数据包，而不必关心发送了什么。不过，如果我们真的需要了解发送的内容，也可以看看DHCP包中租用（或者续租）的IP地址，然后把它也加入我们的网络知识中。

这个程序确实简单，不过也有它的设计局限。为了运行更加有效，必须把程序部署在网络中数据包传输最频繁的位置。对于无线网络来说这比较简单，因为网络中的任何位置都能听到大多数的广播^[注3]。而对于配置了交换机的有线网络，这就比较麻烦。最好的解决方法是为程序配置一个镜像端口（或者称为受控端口），以便它能监听网络中所有的数据包。如果你不是网络管理员，那么可能需要和他们协调来完成配置。如果不能做这样的配置，那么还可以通过类似ARP欺骗和包转发这样的手段来实现，不过这涉嫌制造欺骗信息，所以在这本书中不会介绍^[注4]。

因为我们已经一再地提到网络管理员，所以现在是时候介绍另外一种被动方式的主机发现方法。不过，把这个方法归类到被动方式有些问题，因为它仍然需要我们主动发起查询（只不过是间接查询）。我们在这里发送的数据包不是直接给那些网络中的主机，而是给与它们连接的网络设备。一般可以向这些设备发出三种查询：

- 你知道哪些主机在使用IP？也就是说，请告诉我你的动态ARP表。
- 你知道哪些主机的以太网地址？也就是说，请告诉我你的CAM表。

注3： 假设你已经获得了正确的WEP/WPA密码，成功接入了无线网络。

注4： 如果你希望更加深入了解这个主题，请从<http://monkey.org/~dugsong/dsniff/>下载Dug Song的dsniff软件包。

- 你还知道什么其他的网络设备吗^[注5]?

在第12章的“从Perl中使用SNMP”那一节，我们讨论了如何使用SNMP来获取某台设备的ARP表和动态CAM表（也就是从设备端口采集到的以太网地址表）。修改那一章的脚本应该不难，所以我们就不再介绍相关的代码了。

警告： 这里的提醒在第12章的那一节也提到过，为的是同样的目的。请注意大多数交换机都有学习能力，所以它们的动态CAM表时刻在变化，那些不常用的地址很快会被忘记。这意味着你的程序要反复运行（最起码要比标准的地址淘汰机制更频繁）才能采集到信息。思科设备默认的淘汰周期是3分钟。

第三个问题很容易回答，但也是要看你的网络设备如何配置而决定。很多的设备制造商为自己的产品集成了某种协议，以便它们彼此能发现对方。比如思科发现协议（Cisco Discovery Protocol, CDP）就是比较常见的例子，另外还有Foundry Discover Protocol、SynOptics Network Management Protocol（用于SynOptics、Bay和Nortel的设备）。这时，第三个问题的解答浮出了水面。只要网络环境中配置了以上任何一种协议，我们就可以采用两种方法之一来获取相关设备信息：要么使用Michael Chapman写的Net::CDP（或者类似的专用模块），要么使用Max Baker写的SNMP::Info（或者类似的通用平台）。我个人倾向于使用后者，因为这能使我免于适应一种新的模块。下面的范例代码节选自SNMP::Info::CDP的说明文档，展示了如何查询某台网络设备周边的设备：

```
use SNMP::Info::CDP;

my $cdp = new SNMP::Info (
    AutoSpecify => 1,
    Debug      => 1,
    DestHost   => 'router',
    Community  => 'public',
    Version    => 2
);

my $interfaces = $cdp->interfaces();
my $c_if       = $cdp->c_if();
my $c_ip       = $cdp->c_ip();
my $c_port     = $cdp->c_port();

foreach my $cdp_key (keys %$c_ip){
    my $iid      = $c_if->{$cdp_key};
    my $port     = $interfaces->{$iid};
    my $neighbor = $c_ip->{$cdp_key};
    my $neighbor_port = $c_port->{$cdp_key};
```

注5： 这个问题对于那些大型的网络系统有意义，因为它们充满了变化和未知因素。对于固定的小型网络来说，它们并不值得做这样的努力。

```

    print "Port : $port connected to $neighbor / $neighbor_port\n";
}

```

现在的问题是，如果网络设备不能使用这类发现协议怎么办？这是非常可能的情况，因为从安全角度出发往往有必要关闭这类协议，这样可以使得系统更加“稳固”。如果任由CDP之类的协议在网络中用明文传送路由器拓扑信息，那么任何不熟悉网络结构的家伙就可以借此获悉中心路由设备的位置。在这种情况下，我们得多花些力气才能得到必要的信息。

为了在不能轻易获得设备信息的环境下发现其他网络设备，我们需要获得更高级别的路由信息^[注6]。我们可以从已经找到的设备的BGP、OSPF、RIPvN或者静态路由条目来定位其他路由器，而这只需要我们使用snmpwalk来查询并过滤出重要信息即可：

```

use SNMP;

my $c = new SNMP::Session(DestHost => 'router',
                           Version  => '2c',
                           Community => 'secret');

my $routetable = $c->gettable('ipRouteTable');

for my $dest (keys %$routetable){
    # 3 表示“直接”路由（其他取值的含义参见 RFC 1213）
    next unless $routetable->{$dest}->{ipRouteType} == 3;
    print "$routetable->{$dest}->{ipRouteNextHop}\n";
}

```

到目前为止介绍的内容已经足够我们进一步获得其他信息了^[注7]。

发现网络服务

可以说目前我们已经解决了最复杂的难题，不过下面我们还需要彻底解决遗留的细节问题。现在我们要查找网络中运行了哪些服务，我们可以使用类似IO::Socket或Spider Boardman的Net::UDP这样的模块来向TCP和UDP端口发出连接请求。但这种方法显得过于麻烦，也缺乏技术魅力。

为了扫描网络来发现可用端口，大多数人会选择使用Fyodor编写的nmap (<http://nmap.org>)。这个工具为了速度和性能而进行了充分的优化，目前大概还没有方法能和它竞

注6： 我们还可以考虑主干端口，不过问题是并没有很好的方法来获取主干端口另一头的网络设备的IP地址。如果CDP已经打开，我们就能从cdpCacheTable获得此信息。不过现在它并没有打开，所以只能另寻其他途径。

注7： 这只是冰山的一角。本章末尾列出的Michal Zalewski的书是这方面的入门指南，但其他资料（比如DHCP租约文件，以及IMAP、NetBIOS请求的服务日志文件等等）也很有用。

争，所以用Perl驱动*nmap*应该是最好的选择。调用*nmap*的最强模块应该是Max Schubert的*Nmap::Scanner*。如果你需要分析XML模式的*nmap*输出，那么Anthony G. Persaud的*Nmap::Parser*是最合适的工具。如果你要开始一个这样的项目，请先看看这两个模块。

*Nmap::Scanner*有两种可选的调用模式：批处理和事件驱动。在批处理模式中，我们可以告诉它需要扫描的范围，并且等待它完成之后返回结果。在事件驱动模式中，我们可以注册一个回调函数，它在特定事件发生时（或者发生之前、发生之后）得到运行的机会。比如我们可以让它在每次发现可用端口的时候执行特殊的动作（记录到数据库、发出铃声警报，甚至可以出动飞虎队进行营救）。下面的代码演示了这两种调用模式，先是批处理模式：

```
use Nmap::Scanner;

my $nscan = new Nmap::Scanner;

# nmap 命令的位置。为避免混淆我们这里直接写明路径，
# 但也可以略去不写，它会从环境变量 $PATH 定义的路径中查找。
$nscan->nmap_location('/usr/local/bin/nmap');

# 扫描 192.168.0.x 子网，看看有哪些打开了 80 (http) 端口
my $nres = $nscan->scan('-p 80 192.168.0.0/24');

# 取得符合搜索条件的主机对象列表
my $nhosts = $nres->get_host_list();

# 逐个遍历找到的主机，打印 80 端口开放的主机名
while( my $host = $nhosts->get_next() ){
    print $host->hostname()."\n" if
        $host->get_port("tcp",80)->state() eq 'open';
}
```

现在把代码修改成使用事件模式：

```
use Nmap::Scanner;

my $nscan = new Nmap::Scanner;

$nscan->nmap_location('/sw/bin/nmap');

# 每次发现一个端口，就运行 &PrintIfOpen
$nscan->register_port_found_event( \&PrintIfOpen );

my $nres = $nscan->scan('-p 80 129.10.116.0/24');

sub PrintIfOpen {
    # 每次回调函数被调用时，我们会依次得到
    # 扫描对象、主机对象以及端口对象
    my ( $self, $host, $port ) = @_;
```

```
print $host->hostname() . "\n"
if $port->state() eq 'open';
}
```

在使用*nmap*的时候，我们还获得了额外的好处：自动识别版本和操作系统。默认情况下，*nmap*只会识别可用端口，但是端口并不意味着可用的服务，比如22端口并不总是代表SSH服务。如果我们把代码中的：

```
my $nres = $nscan->scan('-p 80 192.168.0.0/24');
```

改成：

```
my $nres = $nscan->scan('-p 80 -sV 192.168.0.0/24');
```

那么*nmap*会进一步测试打开的端口是否可以提供服务。另外，如果它发现了提供SSL或者TLS服务的端口，它会调用OpenSSL的客户端例程并检查加密端口上提供的服务。真的太棒了！

注意：我们的代码还没有对使用-sV标志返回的信息做出反应。要获取这些信息，我们只需要这样做：

```
$host->get_port('tcp',80)->service->extrainfo()
$host->get_port('tcp',80)->service->product()
$host->get_port('tcp',80)->service->version()
```

给这个脚本加入操作系统判断也很容易：只要加上-o参数就可以了，另外还要拿掉那些端口检查的参数（留着也可以，*nmap*至少需要一个端口才能进行检查），脚本必须以提升权限运行。

物理定位

这个问题可以说是网络发现的极限。每个人都希望能探测出某台主机的物理位置，可是这也是最困难的事情。当一台电脑感染病毒的时候，病毒可能会从网络开始蔓延，这时候你确实（也应该）可以关闭它的网络端口，但是真正需要做的是找到那台被感染的电脑并杀掉病毒。完全依赖于关闭网络端口并不可行，因为用户可能会把电脑连到另一个网络端口，然后一切又会重新开始。

在一些特殊的情况下这个任务变得非常困难。无线网络是最典型的情况，不过甚至在有线网络中也有特殊情况。绝大多数的网络都会使用配线架来把网络设备连到墙上的网络

端口。虽然交换机可以检查配置情况，但是一般来说无法直接查询通往交换机的配线架端口号（除非你买了非常昂贵的配线架），唯一的办法是去楼层配线间看看^[注8]。

不幸的是，目前并没有任何解决此问题的技术方案。不过还是有些技术上的方法可以避免问题。下面列出的方法可能会对你有些启发。

方法一：多方采集信息

如果你能找到这台机器所在位置附近的网络设备，那么就更容易找到它。可以采用下面的步骤：

1. 逐渐把搜索范围缩小到那些最近曾经和那台主机打交道的设备附近。在第12章中，我们看到了如何用SNMP来查询交换机的动态CAM表。如果你的无线接入点支持SNMP协议，那么就可以查询问题主机的MAC地址。通常来说，这样你就可以显著地缩小检查范围。
2. 一旦你能缩小搜索的范围，那么你就可以利用一些常识来找到它。比如无线接入点是典型的辐射中心，因为所有的用户应该都分布在附近的位置^[注9]（具体情况要看无线网络布局和天线架设的位置）。

对于有线网络来说，有时可以使用某台服务器作为搜索的起点。比如，如果你能识别配线架上的服务器，那么这个信息对搜索会有帮助^[注10]。最后，你还可以从打印服务器的日志入手来分析主机在网络中的位置，因为大多数人都会到最近的打印机打印。如果你能找出那台电脑最近使用过的打印机，可能会对找到主机的位置也会有很大的帮助。

方法二：养成好习惯

如果你的网络有命名和连线的惯例，那么它也会对识别问题主机有帮助，可以采用如下步骤：

1. 采用上一节的步骤来逐步缩小检查范围。
2. 如果你的网络有连线规则，那么可以用它来识别问题。你的配线架是否有接线规

注8： 有时候甚至这样也不能奏效。虽然详细跟踪并记录每一个配线架很有用，但是我还从来没有遇到过一个网络管理员未曾需要拔插电缆线的。

注9： 不过这只是意味着大约4到8米的三维空间（大小由天线功率决定）中所有可能的位置。相邻的楼层（甚至是相邻的建筑）都是可能要找的位置。

注10： 可以想象如下的情况：感染病毒的电脑在交换机的5/11端口，而你恰巧知道邮件服务器所在的机房在交换机的5/5端口。如果你的配线架不是很乱的话，应该可以怀疑那台电脑恰好也在那个机房。

则？你是否按照房间编号来命名接口？是否可以使用反向DNS查询来获得主机所在的网段？

我知道这些方法都不算完美，但是希望它能帮你开始认识问题。

展现信息

在开发网络监控程序的时候，有四个必要的部分：数据获取（也就是探测）、数据展现、控制框架以及最终的分析和警告。之前我们已经花了不少时间讨论了第一个部分，所以让我们进入下一个主题。

文本展现工具

在好莱坞大片中，网络监控程序无一例外有漂亮的图形界面、超大地图显示器和闪烁的警告灯，有时候还有非常能制造科幻气氛的滴滴声。不过在现实中，这些却并非最常见的网络状况展现手段，因为有很多信息是用文本格式来展现的。纯文本便于用邮件发送，也能用智能手机阅读，还是绝大多数系统支持的格式。

这一节我们会展示那些能够帮助我们呈现信息的Perl工具。虽然这里我们主要是用它们来进行网络监控，但这些工具其实可以在任何需要文本输出的场合中使用。请尽可能多地使用它们，相信你会越来越喜爱这些工具。

注意：要记住Perl有众多值得一提的工具，我们并不会一一介绍。这些工具软件包和模块专注于支持常规的模板工作，它们为此定义了很多可以插入文档中的标记（或者小型语言），从而实现报表和网页的生成，以及许多其他用途的输出。这些工具包括Template Toolkit、Text::Template和HTML::Template。

这些工具能完成我前面提到过的所有任务（大多数的工具都支持内嵌的Perl代码，所以它们实际上能实现所有Perl能做到的事情），但是这些工具往往太复杂，会给原本简单的任务带来没有必要的麻烦。所以我们会介绍那些简单、功能单一的工具。对于更麻烦的任务来说，请尽可能使用这些大型软件包和模块，因为它们就是为此而生的。

让我们先来看看那些能让文本显示得更加专业的工具。有众多的格式化工具能重定文本格式^[注11]。这种重定格式（reformatting）意味着在必要的时候进行折行、删除不必要的空格和标点符号、进行大小写转换等等。这一类的模块包括David Muir Sharnoff的Text::Wrap和José Alves de Castro的Text::Beautify。另外，我发现自己开始越来越多地使用Damian Conway的Text::Autoformat。这个模块经过精心设计，能轻易实现人工

注11：这里大多数模块都能让英文（或者相似的文字）显示得更加专业。我不太确定这些模块对于其他非英文文本会如何工作。

格式化的大多数效果，它能识别缩进，支持列表格式和各种引用惯例。这些功能都可以配置，不过模块很少需要花这些工夫。

Text::Autoformat在使用上的简洁是可圈可点的：

```
use Text::Autoformat;
my $a= 'This is an example of really long text that blathers on and on.
      Strangely formatted, too.
      Should it be presented to
      a user in this form? Probably not.
      Here are three good reasons:
      1) we really don't want our lists to look bad. Ideally we'd
      like the numbered lists to wrap properly too.
      2) it looks unprofessional
      3) we need an example
      ';
print autoformat ($a, {all=>1});
```

这段程序的效果是：

```
This is an example of really long text that blathers on and on.
Strangely formatted, too. Should it be presented to a user in this form?
Probably not. Here are three good reasons:
  1) we really don't want our lists to look bad. Ideally we'd like the
    numbered lists to wrap properly too.
  2) it looks unprofessional
  3) we need an example
```

这段漂亮的输出就是一个autoformat()调用产生的。这里唯一不太容易看懂的代码是一个可选参数：默认情况下autoformat()只会对第一段文本进行格式化，必须加上all参数才能进行整篇文章的格式化。

注意：请务必使用Text::Autoformat的最新版本（写作本书的时候是1.14.0版），因为在处理行末的冒号时曾经有些问题，而我们的范例恰好有这个情况。

现在我们已经展示了格式化文本的简单方法，下面我们就可以看看其他格式的数据展现。列和表格是最常见的格式，在通过邮件展现列表的时候，我们常常会使用列对齐的格式。Alan K. Stebbens的Array::PrintCols非常擅长完成此类任务。比如这段代码：

```
use Array::PrintCols;

my @a = ('Martin Balsam','John Fiedler','Lee J. Cobb','E.G. Marshall',
        'Jack Klugman','Ed Binns','Jack Warden','Henry Fonda',
        'Joseph Sweeney','Ed Begley','George Voskovec','Robert Webber');

$array::PrintCols::PreSorted = 0; # 原始数据没有预先排序，所以现在需要排序
print_cols \@a;
```

会产生下面的输出：

E.G. Marshall	George Voskovec	Jack Warden	Lee J. Cobb
Ed Begley	Henry Fonda	John Fiedler	Martin Balsam
Ed Binns	Jack Klugman	Joseph Sweeney	Robert Webber

`Array::PrintCols`还能被配制为只打印几列（或者调整列宽），请参考它的说明文档。

在掌握了列表的处理方法之后，下一个问题就是如何处理表格。下面的范例输出并不难生成：

```
+-----+-----+-----+
| Host   | Status | Owner |
+-----+-----+-----+
| brady  | passed | fmarch |
| drummond | passed | stracy |
| hornbeck | passed | gkelly |
+-----+-----+-----+
```

不知道什么原因，文本处理得到众多模块开发者的青睐，哪怕这方面已经有了众多的解决方案。所以提起表格的输出，可以选择的模块起码包括`Text::TabularDisplay`（作者是Darren Chamberlain）、`Text::FormatTable`（作者是David Schweikert）、`Text::ASCIITable`（作者是Håkon Nessjøenand）以及`Data::ShowTable`（作者是Alan K. Stebbens）。在这些模块中，我最喜欢的是`Text::FormatTable`，因为它简单易用。

刚才的输出可以从下面的代码产生：

```
use Text::FormatTable;

# 可以假想一下，类似这样的复杂数据可能来自网络探测进程
my %results = (
    'drummond' => {
        status => 'passed',
        owner  => 'stracy'
    },
    'brady' => {
        status => 'passed',
        owner  => 'fmarch'
    },
    'hornbeck' => {
        status => 'passed',
        owner  => 'gkelly'
    }
);

my $table = Text::FormatTable->new('| 1 | 1 | 1 |');
$table->rule('-');
$table->head(qw(Host Status Owner));
$table->rule('-');
```



```

for ( sort keys %results ) {
    $table->row( $_, $results{$_}{status}, $results{$_}{owner} );
}

$table->rule('-');
print $table->render();

```

使用这个模块创建表格只需三步：

1. 创建一个表格对象，并同时声明列的数量和数据对齐的方式（如文本是左对齐还是右对齐）。
2. 填充表格内容，从表头开始。创建表头其实非常简单，只要使用

`\$table->rule('-')`	
来画一条分割线就可以了。对于表格的每一行，只要通过调用`row()`来填充它们就行了。`row()`的每一个参数都是表格的一列。在填充完所有的行之后，再次用同样的方法画一条分割线。	
3. 通过`render()`调用来生成表格并打印结果。

本节最后介绍的模块是Kirk Baucom写的Text::BarGraph。通过如下的简单程序：

```

use Text::BarGraph;

# 把这些想象为搜集自每台主机的重要运行状态数据
my %hoststats = ( 'click' => 100,
                  'clack'  => 37,
                  'moo'    => 75,
                  'giggle' => 10,
                  'duck'   => 150);

my $g = Text::BarGraph->new();

$g->{columns} = 70; # 设置列宽
$g->{num}      = 1; # 在条形边上显示数值

print $g->graph(\%hoststats);

```

我们可以产生如下的字符条形图：

```

clack ( 37) #####
click (100) #####
duck (150) #####
giggle ( 10) ###
moo ( 75) #####

```

这样的图形报表可以很容易以邮件方式发送。

图形展现工具

上一节展现的文本图是很好的开始，现在可以进一步讨论如何用真正的图形来展现信息。

使用GD::Graph系列模块

当人们想到信息的展现时，往往会联想到漂亮的图形。Perl有众多产生图形的方式，包括调用外部绘图程序（比如*gnuplot*和*ploticus*）或者通用的数字捣弄器（如Excel和Matlab），还包括复杂的三维OpenGL图形生成器。其中最简单（也最直接）的创建图形的方法就是使用Martien Verbruggen的GD::Graph系列模块（包括其他人贡献的插件）。

注意：不过我们要先提醒一下，这个系列的模块并不容易构造，因为GD::Graph模块依赖于Lincoln Stein的GD模块，而GD模块依赖于Thomas Boutell的GD库（C语言库）。GD库又至少依赖于另外五个C语言库：zlib、libpng、FreeType、JPEG和XPM，而这些模块又有自己的依赖库。现在你大概明白我的意思了吧，所以有人把安装这个模块比喻成是给牦牛剪毛。虽然自己动手做也值得，但是如果你能使用预先编译好的模块，那又何乐而不为呢？

使用这些模块的第一步是选择图形的类型。考虑到可选的数量之多，这并不是一件容易的事情。在写作本书的时候，可选的图形包括面积图、条形图（横向或纵向）、直方图、折线图、点状折线图、圆饼图、波形图，还有时间表图。

一旦你选择了合适的图形类型，制图的过程就显得很简单了：

1. 载入合适的子模块。比如对于纵向条形图来说，我们可以使用GD::Graph::hbars：

```
use GD::Graph::hbars;
```

2. 确保数据的格式正确。数据接口的格式至少包括两个数组引用。第一个数组引用应该指向一个标签值的列表，其中的字符串会成为图形的X轴（横轴），对于圆饼图来说也就是切片的名字。第二个数组引用则描述Y轴（纵轴）的值，也就是饼图的切片大小：

```
my @data=(["click clack moo giggle duck"],[100,37,75,10,150]);
```

3. 创建一个这种图形类型的对象：

```
my $g = new GD::Graph::hbars;
```

4. 画出图形：

```
$g->plot(\@data);
```

5. 写到磁盘：

```
open my $T, '>', 't.png' or die "Can't open t.png:$!\n";
binmode $T;
print $T $g->gd->png;
close $T;
```

这段代码中除了关于binmode的部分之外都很好懂。这个命令对于Perl来说比较少用，所以我们只有在遇到问题时才会想到它。简单说来，它能确保在那些区分I/O处理模式的操作系统上正常工作。最常见的应用场合是Windows Perl编程，因为Windows操作系统需要明确区分文本和二进制模式。

图13-2展现了我们的代码的运行结果。

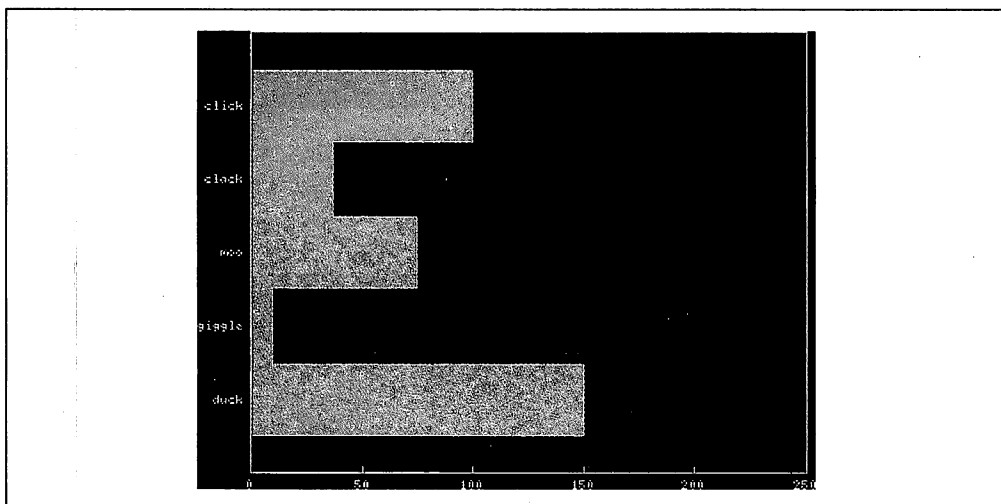


图13-2：范例程序产生的图形

看上去有些太简单了，不是吗？其实图形简单并不是大问题（请看本章末尾部分参考资料中Edward Tufte对“图形垃圾”的驳斥），但是我们确实还能让图形更加好看一些。要改善图形显示效果，可以在创建图形对象之后设置一些选项，然后再完成数据制图：

```
$g->set(
    x_label      => 'Machine Name',
    y_label      => 'Bogomips',
    title        => 'Machine Computation Comparison',
    x_label_position => 0.5,
    bar_spacing  => 10,
    values_space  => 15,
    shadow_depth  => 4,
    shadowclr     => 'dred',
    transparent   => 0,
    show_values   => $g
);
```

现在我们获得了如图13-3所示的结果。

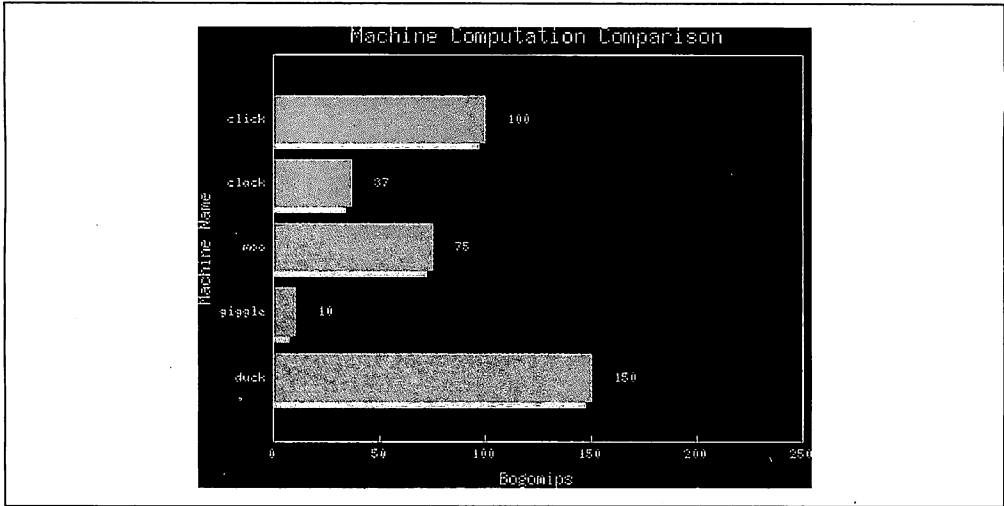


图13-3：以格式化改善后的图形

现在好多了。不过仍然有很多方法可以改善图形的显示。如果你的目标就是要让图形尽可能给人留下深刻印象，那么还可以使用Jeremy Wadsack的GD::Graph3d模块。如果我们把代码中的hbars改成bars3d，那么就会获得如图13-4显示的效果。

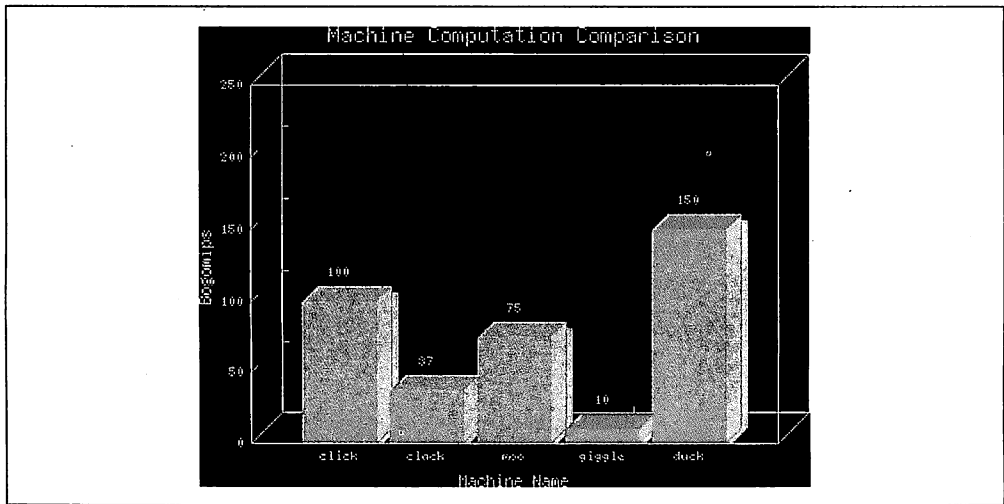


图13-4：图形的3D版本

使用GraphViz

图表并非我们唯一可以使用的展现信息途径。我曾经见过迫切需要网络拓扑图来说明网

络结构的情况。AT&T公司的GraphViz可视化软件 (<http://www.graphviz.org>) 可以通过 Leon Brocard的GraphViz模块来驱动, 这正是我们需要的工具。

图13-5显示了一个使用GraphViz绘制的简单结构图。

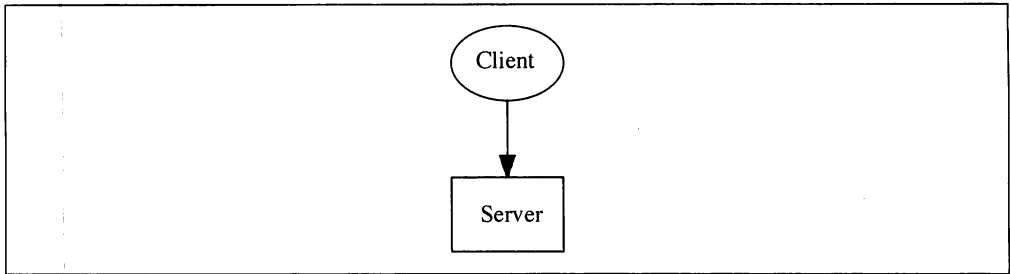


图13-5: GraphViz结构图

用来产生这个图的代码很简单:

```
use GraphViz;

my $g = GraphViz->new();
$g->add_node('Client');
$g->add_node('Server', shape=>'box');
$g->add_edge('Client' => 'Server');

$g->as_jpeg('simple.jpg');
```

创建对象, 然后加入两个节点, 加上它们之间的连接器, 把文件保存为JPEG文件, 这真的是非常简单。这个例子过于简单, 还没有展现出GraphViz模块的真正实力。对于更加复杂的图表, 这个软件包的计算能力才能充分显现出来, 因为它生成的图表可以说是非常美观。如果采用其他的软件包, 你可能要花很多时间来调整它生成图表的布局, 而GraphViz就能替你省去那些力气。

让我们用一个更加复杂的例子来测试这个工具。下面的代码能嗅探发往HTTP端口的前50个SYN数据包, 并且用哈希保存它们的HTTP源地址和目标地址。然后使用几个GraphViz命令来制图, 这样我们能看到哪些机器正在访问我们的网站。这里的代码和前面相比只是多加了几个命令, 所以应该很容易理解:

```
use NetPacket::Ethernet qw(:strip);
use NetPacket::IP qw(:strip);
use NetPacket::TCP;
use Net::PcapUtils;
use GraphViz;

my $filt = 'port 80 and tcp[13] = 2';
my $dev = 'en1';
my %traffic; # 用于记录源地址-目标地址对
```

```

die 'Unable to perform capture: '
. Net::Pcap::geterr($dev) . "\n"
if ( Net::PcapUtils::loop(
    \&grabipandlog,
    DEV      => $dev,
    FILTER    => $filt,
    NUMPACKETS => 50 )
);

my $g = new GraphViz;

for ( keys %traffic ) {
    my ( $src, $dest ) = split(/:/);
    $g->add_node($src);
    $g->add_node($dest);
    $g->add_edge( $src => $dest );
}

$g->as_jpeg('syn80.jpg');

sub grabipandlog {
    my ( $arg, $hdr, $pkt ) = @_;

    my $src = NetPacket::IP->decode( NetPacket::Ethernet::strip($pkt) )
        ->{'src_ip'};

    my $dst = NetPacket::IP->decode( NetPacket::Ethernet::strip($pkt) )
        ->{'dest_ip'};

    $traffic{"$src:$dst"}++;
}

```

范例的输出如图13-6所示。

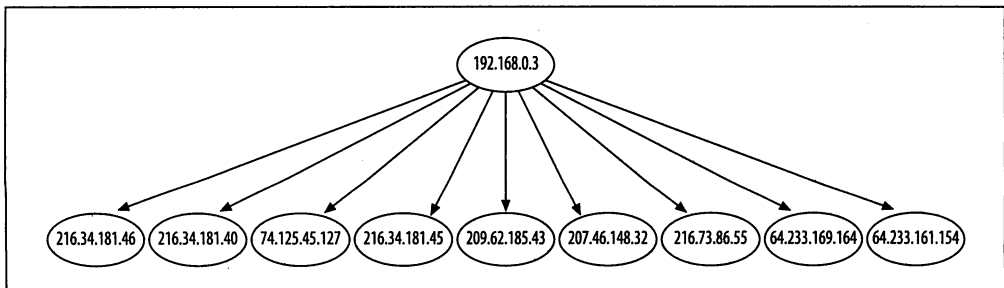


图13-6：默认输出

以上的输出是默认格式，如果我们把下面这行：

```
my $g = new GraphViz;
```

改成：

```
my $g = new GraphViz(layout => 'circo');
```

那么可以产生如图13-7所示的效果。

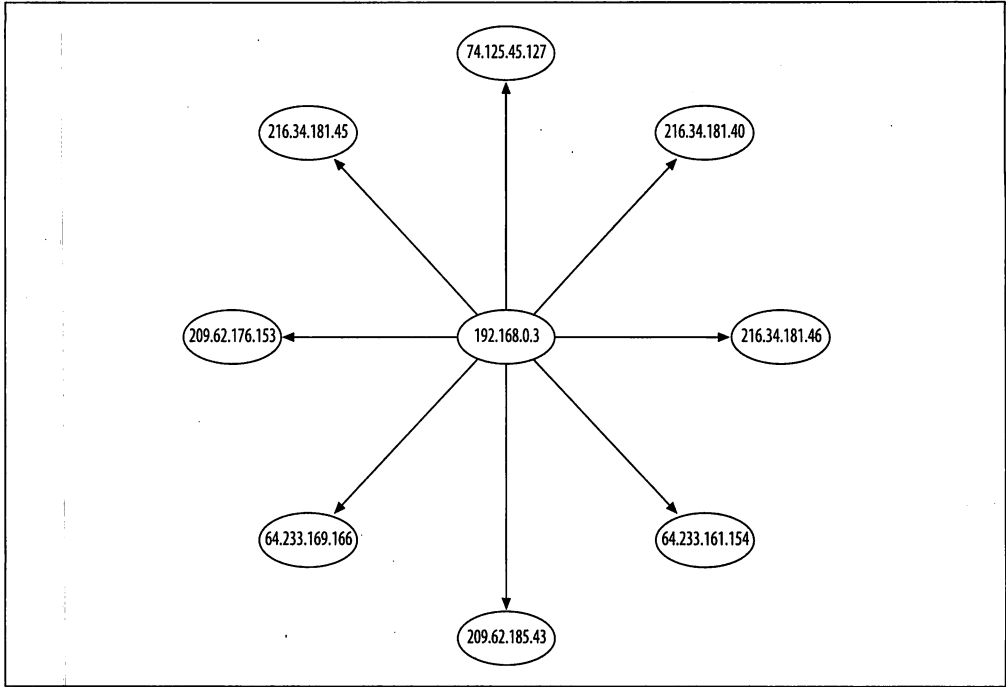


图13-7：“circo”布局的输出

而把那一行改成这样：

```
my $g = new GraphViz(layout => 'fdp');
```

又能产生如图13-8所示的效果。

GraphViz有众多的格式化选项，请参考说明文档来了解详细信息。

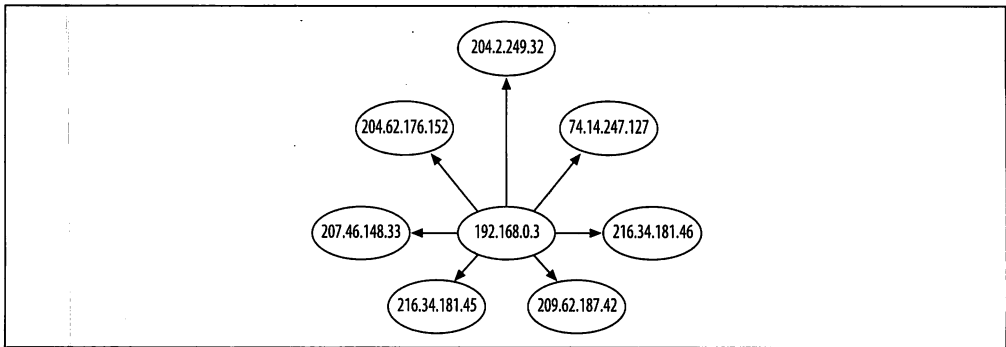


图13-8：“fdp”布局的输出

注意：顺便说一下，还有一个值得尝试的模块系列`Graph::Easy`，作者是Tels。相比于`GraphViz`来说，它的调用接口更加简单。另外它还有一些很独特的功能，比如ASCII图表输出。

在结束关于`GraphViz`的讨论之前，还要提醒你去关注那些基于`GraphViz`的衍生模块。即使不使用衍生模块，`GraphViz`本身也带有一些子模块，用来为Perl数据结构、正则表达式、解析语法和XML文件生成图表。而衍生的模块可以通过DBI为数据库绘制结构图，为`Makefile`生成依赖图，为邮件主题生成序列图，还有DNS区域图等等。所以没有办法预测你会发现什么样的`GraphViz`应用。

使用RRDtool

我把RRDtool这种图形展示工具留到最后介绍是因为它最为复杂。我们只会介绍这种工具的一些简单应用，相信你在理解之后能找到更多深入应用的机会。

人们常常发现刚开始使用RRDtool时会有些让人摸不着头脑，这是因为大家往往错误地把它理解成了另外一种软件。下面我们就先清除两种关于这个工具的误解，这样你就不会落入迷雾之中：

- RRDtool并非一种通用的绘图软件包。绝大多数情况下你不能指望通过输入数据就立刻得到那种有X-Y轴的图形。如果你需要的是这样的软件，那么应该去找前面介绍过的`GD::Graph`系列模块。
- RRDtool也不是一种常用的数据库工具，比如典型的Berkeley DB或者{s,n,g}bm数据库。如果你的应用程序需要那种存储模型，可以看看BerkeleyDB或者`DBM::Deep`模块。

在开始学习RRDtool的时候，最关键的是要清楚理解这个工具的概念。如果你把它理解成绘图软件或者关系数据库，那么这是一个错误的开端。所以我们这里会先阐明RRDtool的核心概念，然后再介绍它的命令行调用方法。

要掌握RRDtool的概念，一个好的开始是构思合适的应用场景。典型的RRDtool应用是路由器的监控。一般来说路由器有很多的信息可以监控，但是我们会以三个指标用于这个范例：流入带宽、流出带宽和路由器温度。为了跟踪这三个指标的波动，我们会每两分钟查询一次。这个频率对于绘制路由器的监控图来说应该足够了。

为了实现这个目标，我们需要理解五个基础的RRDtool概念：

RRDtool基本原理一：定时提供数据

RRDtool的数据库可以理解成以时间为刻度的保存数据的时间桶（time bucket），每个刻度都对应于一段定长的时间片，数据库希望你在其中存放相应时间的数据。不过在现实世界中，很难保证非常精确地定时采集数据。所以RRDtool会自动帮你

分析采集到的数据所在的时间片并存储到相应的位置去。另外，RRDtool还要求你的数据是连续的且可计算的。

我之所以使用“时间桶”这个比喻（而不是“时间槽”）是因为RRDtool能够处理好采集周期的波动问题。比如一个设置为每5分钟采集一次的数据库可以对第6分钟采集的数据做自动的平均值处理。这可以想象成一个老式的奶桶，顶一般有个漏斗，漏斗能够很好地把奶接到桶口里去。这非常类似RRDtool的工作机制。

RRDtool基础二：未知数据

用户需要采集每个时间片对应的数据，以此更新数据库。如果你不能提供某个时间片的数据，那么就会产生未知数据。如果某个桶里面超过一半的数据是未知的，那么整桶数据就会被标记为*UNKNOWN*。

设立*UNKNOWN*数据主要是为了进一步计算（比如日平均值）的需要。RRDtool可以对某种计算需要的已知值和未知值的比例进行配置，如果超过一定比例的值是未知的，那么整个计算的结果就会被标记为*UNKNOWN*。这有些类似于电子表格，其中某些格子中的数据错误可以导致整个公式的计算错误。

RRDtool基础三：有限数量的时间桶

在创建RRDtool数据库的时候，预先定义了时间桶的数量（足够盛放一段时间内采集的数据即可）。RRDtool采用了和我们在第10章中看到的bigbuffy程序一样的循环缓冲方法。从数据库的头部开始记录数据，直到到达数据库的结尾，然后回到头部重新记录。这正是这个工具被称为“round robin database”的原因所在。这个循环缓冲方式使得RRDtool能够充分使用空间来存放某段时间内足够的数据。

RRDtool基础四：主要数据点（PDP）和累计数据点（CDP）

RRDtool收到新采集的数据时，会进行两个操作：在数据库中记录主要数据点（也就是实际采集的数据）；同时记录累计数据点（也就是从这个数据和其他同一桶内PDP计算出来的值）。比如RRDtool被配置为记录某个指标的每小时平均值，那么它会在数据库中同时存储新采集的指标值（PDP）和桶内每小时平均值（CDP）。所以需要记录CDP是因为它往往是实际输出的图形中的值。RRDtool对输入数据进行这样的处理是为了使得平均值的计算更加精确。

这个特点显示出RRDtool和其他数据库的差异。因为大多数数据库都是用来获取和输入数据一致的值，而RRDtool不是这样。对于我们的应用来说，需要绘制的是路由器的平均输入带宽的时间波动图，而带宽的趋势是一种需要用时间片来观察的数据，所以实际采集到的输入字节数对我们来说并没有太大的意义。

RRDtool基础五：数据源类型（DST）

在创建RRDtool数据库的时候，你可以选择它用来展现的数据类型。这个类型决定

了将要采集的数据的结构。可选的范围包括counter、derive、absolute、gauge和compute。

注意：马上就会介绍这一节最重要的一个词汇：“变化率”（rate of change）。

RRDtool设计时最关心的是如何展现数据随时间变化的情况。虽然你确实可以使用RRDtool来直接存放原始数据，但是它真正感兴趣的是显示数据的“变化率”。所以你最好在开始的时候就牢牢记住“变化率，随时间变化，变化率……”这句话。这会帮助你正确设计RRDtool存储和展现的数据。

counter是最常用的DST。它用来存放递增的计数器的变化率。大多数的计数器都有自己的上限，达到之后会清零，路由器就有很多这样的指标。RRDtool能妥善处理这种情况：如果传递给RRDtool的counter指标的PDP值比之前的那个值小，那么它就判定计数器发生了指标回绕，所以新的指标实际上是上一个值到回绕上限的差异再加上当前的值。不过这个counter数据类型的功能有时候也会因为设备复位或重启导致的指标清零而受到干扰。要知道，如果没有设置指标的上限和下限，RRDtool没有办法区分回绕和复位清零，所以它只能假定所有的情况都是回绕。如果你不能接受这个问题，那么还有两个解决办法：你可以把某个值手动设置为“U”（也就是未知）来避免RRDtool进行错误的判断；或者，你还可以使用derive数据类型，并且把下限设置为0。

derive类似于counter，只是没有回绕的逻辑。说得更加详细点，（根据文档介绍）它是“从上一个数据到当前数据的线性导数”。不过你不必立刻回去翻线性代数的书，它的意思其实是说derive数据类型能够处理递增和递减两种变动率。例如，可用磁盘空间就是一个典型的可增也可减的指标，既可以因为磁盘空间清理而腾出更多可用空间，也可以因为使用而减少。

absolute往往用来表述那些每次采集后都会自动复位的计数器。这个类型很少见，所以我觉得使用这个类型就是对RRDtool说：“其实只要把我给你的数据除以时间片的长度，那就是变动率了。”比如，假定我们要监视邮件服务器拒绝的病毒附件的趋势，你可能需要每5分钟采集一次被拒绝的邮件数量，如果最近一次采集的结果是30，那么对于absolute数据类型来说变动率就是 $30 / (5 * 60 \text{ 秒})$ 。

gauge可能是第二常见的数据类型。与其他数据类型不同，gauge值不需要任何计算。它们不必进行变动率的推算，因为它们本身就是监视所需要的变动趋势。所以如果在上一个例子中你关心的就是被拒绝的邮件的数量（而不是它的变动情况），那么你就可以使用gauge类型。

compute这个类型在RRDtool的指南当中没有详细介绍，因为这可能会吓倒初学者。RRDtool对某些计算会使用逆波兰表达式(Reverse Polish Notation, RPN)来编

程。在这里，逆波兰表达式会用来指示要如何从其他的数据源来推算某个数据源。这有些类似于使用公式来推算某个电子表格单元的数值（只不过需要使用逆波兰表达式）。如果你觉得自己在计算器上曾经操练过的逆波兰表达式已经有些遗忘了，请查看RRDtool附带的专题指南。

好了，理论课就到此为止。现在让我们看看要如何用这些知识来创建RRDtool数据库、传入数据并产生图形吧。首先是数据库的创建：

```
$ rrdtool create router.rrd --start `perl -e 'print time-1` \
    --step 120 \
    DS:bandin:COUNTER:240:0:10000000 \
    DS:bandout:COUNTER:240:0:10000000 \
    DS:temp_in:GAUGE:240:0:100 \
    RRA:AVERAGE:0.5:30:24
```

让我们分段来阅读上面的命令。首先，我们以当前时间为起始点创建数据库，并且声明数据需要每2分钟（也就是120秒）采集一次：

```
$ rrdtool create router.rrd --start `perl -e 'print time-1` \
    --step 120 \
```

我们打算每2分钟采集三组数据（流入带宽、流出带宽和路由器的环境气温）。如果我们在上次采集之后有240秒没有向RRDtool提供新数据，那么那个时间桶会被标记为*UNKNOWN*。带宽数据源的下限和上限分别被指定为0MB和10MB（假定我们监视的是一台10MB带宽的路由器）。这是为了帮助RRDtool来侦测路由器重启。另外，我们还设定环境气温的下限和上限分别是水的冰点和沸点（0-10℃）^[注12]：

```
    DS:bandin:COUNTER:240:0:10000000 \
    DS:bandout:COUNTER:240:0:10000000 \
    DS:temp_in:GAUGE:240:0:100 \
```

最后，我们还需要存储的是一天的累计数据点（CDP），用来表示每小时平均值（每天有24小时，而每小时有30个2分钟）。另外，我们输入的三组指标都需要每小时平均值。而下面的0.5参数则是在“RRDtool基本原理二”部分介绍过的，表示一旦桶里有一半以上的数据是*UNKNOWN*，那么相应的CDP也不必再计算，可以直接标记为*UNKNOWN*：

```
    RRA:AVERAGE:0.5:30:24
```

要知道我们可以保留任意长度的 RRA（Round Robin Archive），比如可以计算月度甚至年度平均值，但是这个例子里面只保留一天的历史数据。

Perl版本的create命令可以说是对命令行的简单翻译：

注12：不过，如果你的数据中心真的达到任何一端的温度，请别邀请我去参观！

```

use RRDs;
my $database = "router.rrd";
RRDs::create ($database, '-start', time-1, '-step', '120',
               'DS:bandin:COUNTER:240:0:10000000',
               'DS:bandout:COUNTER:240:0:10000000',
               'DS:temp_in:GAUGE:240:0:100',
               'RRA:AVERAGE:0.5:30:24');

my $ERR=RRDs::error;
die "Can't create $database: $ERR\n" if $ERR;

```

有了数据库之后，我们可以开始对指标进行每2分钟的采集：

```

rrdtool update router.rrd N:25336600490171:159512031730187:26
(2 minutes go by)
rrdtool update router.rrd N:25336612743804:159512154231472:26
...
rrdtool update router.rrd N:25336810864361:159513632487313:26
...
rrdtool update router.rrd N:25336950227556:159515045447411:26
...
rrdtool update router.rrd N:25337088963449:159516528948027:26
...
rrdtool update router.rrd N:25337088963449:159516528948027:26
...

```

这里的第一个参数是刚才创建的数据库的名称。然后跟着最新的输入数据。其中第一个字段的N代表Now，也就是当前的系统时间。如果出于某种考虑我们不希望用当前时间作为输入数据的相关值（比如重新计算并载入历史数据），那么可以使用刚才看到的rrdtool create命令中的格式的时间（也就是time()的输出）。时间戳之后的字段是按照我们的rrdtool create或者RRDs::create()规定的顺序排列的数据源的值。

你应该觉得很习惯，Perl版本的数据输入也同样是命令行的简单翻译^[注13]：

```

RRDs::update('router.rrd', 'N:25336600490171:159512031730187:26');
RRDs::update('router.rrd', 'N:25336612743804:159512154231472:26'); ...

```

当然从编程角度来说，你写的代码应该类似这样：

```

while (1) {
    ($in,$out,$temp)= snmpquery(); # 通过 SNMP 查询路由器
    RRDs::update($database, "N:$in:$out:$temp");
    my $ERR=RRDs::error;
}

```

注13：如果你对于这个庞大的范例数据的来源感兴趣，那么我要告诉你，它其实是来自我的思科路由器的真实数据。采集的手段是对1.3.6.1.2.1.31.1.1.1.6.74 (ifHCInOctets)、1.3.6.1.2.1.31.1.1.1.10.74 (ifHCOutOctets) 和1.3.6.1.4.1.9.9.13.1.3.1.3.1 (ciscoEnvMon Temperature StatusValue) 三个OID进行snmpget调用（通过SNMPv2c协议）。这里我没有展示snmpget命令，为的是让大家专注于RRDtool。

```

die "Can't update $database: $ERR\n" if $ERR;
sleep (120 - time % 120);      # 休眠一段时间再继续采集数据
}

```

要知道我们之所以介绍RRDtool是因为它能生成监控图，不过到目前为止还没有看到任何成果。而现在就是享受成果的时间了，我们这就开始展示那些漂亮的图形，会有两幅图（原因马上揭晓）。

RRDtool的绘图功能往往也是初学者会遇到的困难，因为它很容易越变越复杂。所以作为初学者的入门指南，我们只会简单介绍最基本的功能。不过你可以在掌握这些基础之后深入学习RRDtool的文档来了解其他功能。

为了绘制刚才采集的数据的图形，我们需要指定三个基本参数：

- 图形输出文件的名字。可以使用减号 (-) 来表明要输出到标准输出。
- 一个以上的数据定义，这样RRDtool才知道要从数据库中取出的指标值，并进行必要的计算，从而产生图形。
- 图形的描述（也就是绘制的内容）。

现在让我们先绘制路由器的带宽信息。命令行的版本是这样的：

```

rrdtool graph bandw.png \
    DEF:bandwin=router.rrd:bandin:AVERAGE \
    DEF:bandwout=router.rrd:bandout:AVERAGE \
    LINE2:bandwin\#FF0000 \
    LINE2:bandwout\#000000

```

这个命令能产生如图13-9所示的图形。

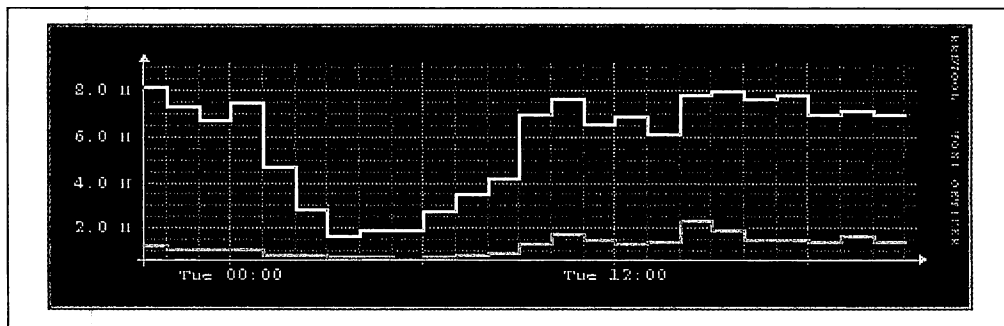


图13-9：路由器带宽波动图

让我们把这个命令分成小段来解释。首先是数据定义：

```

DEF:bandwin=router.rrd:bandin:AVERAGE \
DEF:bandwout=router.rrd:bandout:AVERAGE \

```

这里声明的是需要从数据库router.rrd中获取bandin和bandout指标的平均值，并且用bandwin和bandwout来代表它们：

```
LINE2:bandwin\#FF0000 \
LINE2:bandwout\#000000
```

下面这两行说明需要使用中等粗细的线条（LINE2）来绘制bandwin和bandwout的图形，相应的颜色则用十六进制指定。这并不太复杂，是吧？默认情况下图形会包含一整天的数据，不过我们也可以指定具体的绘图开始和结束绘图时间。为了避免重复，我们会使用Perl代码来生成下午1点到5点的图形：

```
use RRDs;
RRDs::graph('dayband.png',
    '-start', '1234893600', '-end', '1234908000',
    '--lower-limit 0',
    'DEF:bandwin=router.rrd:bandin:AVERAGE',
    'DEF:bandwout=router.rrd:bandout:AVERAGE',
    'LINE2:bandwin#FF0000',
    'LINE2:bandwout#000000');
```

图13-10就是程序的输出。

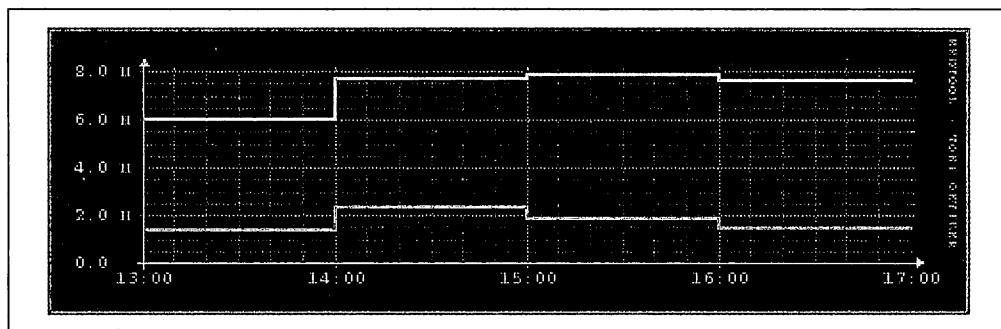


图13-10：4小时的带宽波动图

好了，现在让我们尝试输出温度变化图。如果使用最简洁的代码来完成制图，那么应该是这样的：

```
use RRDs;
RRDs::graph('temp.png',
    'DEF:temp=router.rrd:temp_in:AVERAGE',
    'LINE2:temp#000000');
```

图13-11显示了这段代码的输出。

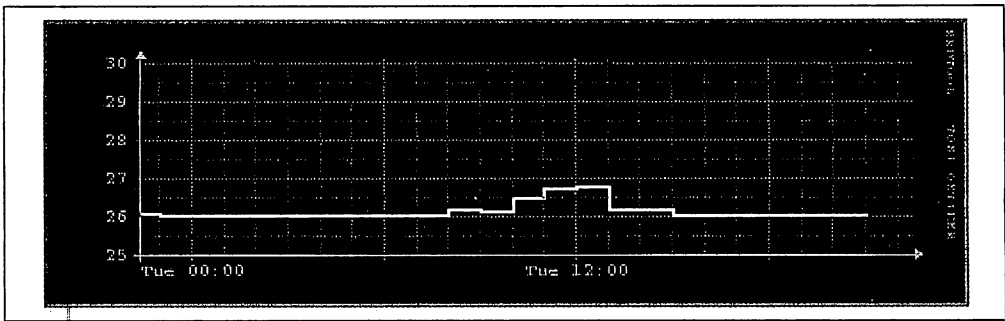


图13-11：温度波动图

关于这个图形有些要补充的说明。首先，它显得非常平淡、呆板。不过这里缺少波动其实是一件好事！因为这个图形反映的是进入路由器的空气的温度^[注14]。如果这个温度上下波动，那么意味着数据中心的冷却系统出了问题。其次，这里的所有温度都保持在26—27℃之间。相比之前的带宽图来说这个图形的粒度太细了，所以我们有必要另外生成一幅温度波动图。

如果我们真的希望改进图形的输出效果，那么有几件可以做的事情。首先大多数美国人都习惯使用摄氏温度，所以我们可以使用RRDtool内置的计算功能来把它换算成华氏温度。另外我们还可以在85°F绘制一根警告线：

```
use RRDs;
RRDs::graph('tempf.png',
    'DEF:temp=router.rrd:temp_in:AVERAGE',
    'CDEF:tempf=temp,9,*,5,/,32,+',
    'LINE2:tempf#000000:Inflow Temp',
    "LINE:85#FF0000:Danger Line\r");
```

现在我们获得了如图13-12所示的输出。

让我们看看刚才的代码有哪些改进。首先，下面的两行是相关的：

```
'CDEF:tempf=temp,9,*,5,/,32,+',
'LINE2:tempf#000000:Inflow Temp',
```

第一行的逆波兰表达式之前曾经提到过。它对RRDtool数据库中的温度数据乘以1.8（也就是9/5），再加上32，换算成华氏温度。然后下一行把这个指标加入图形中，同时也加入图形下方的图标。

注14： 另外还有一个SNMP OID可以用来采集路由器排出的气温（这可以反应路由器内部的电路运行状况，还有更重要的风扇运行情况和通风情况）。它的OID是1.3.6.1.4.1.9.9.13.1.3.1.3.3。

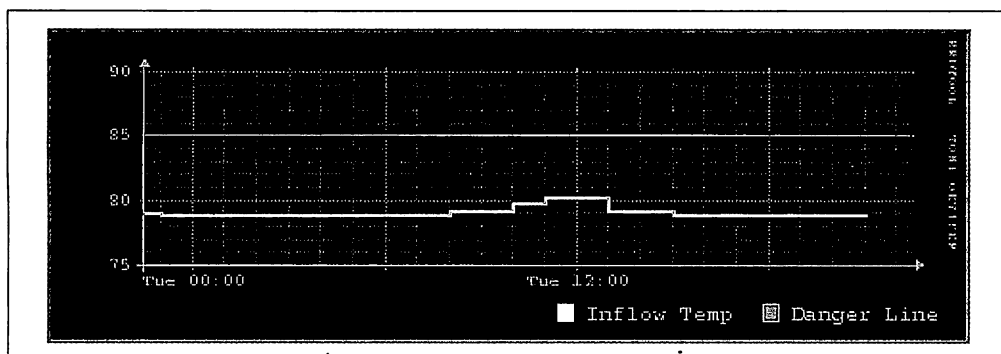


图13-12：华氏温度波动图

其实，还有一行没有见过：

```
"LINE:85#FF0000: Danger Line\r"
```

这一行意味着要在85°F绘制一根直线，并在图的下方加入相应的图标。

我们已经介绍了许多RRDtool应用。RRDtool的功能很多，文档也非常丰富，这是因为它有一个非常负责的开发者 and 活跃的用户社区。希望以上简介能让你顺利开始使用它，也能早日接触到更加深入的应用领域。

监控框架

现在是时候把我们介绍过的内容组合起来，创建一个网络监控的框架了。我们已经讨论了如何获取和展现信息，现在只需要一个东西来驱动整个流程即可。这里我们先尝试用自己搭建的系统来进行监控，然后逐渐过渡到那些现有的更复杂的监控系统。

要知道，我们可以用Perl自带的功能构造一个最简单的监控系统。2003年Randal Schwartz在《Linux Magazine》的某一期上发表了一篇使用Test::More模块来监控网站的文章。Test::More提供了一个用于编写测试脚本的框架，你只要告诉它需要运行的测试和期望的测试结果，它就会帮你完成剩下的工作。

Schwartz的文章描述了如何通过Test::More脚本来连接网站，并且判断它返回的数据是否为期望的内容。很容易就可以把这个例子扩展为监控整个网络的机制。只要你能编写测试来监控网络和主机的运行状况，Test::More就能完成剩下的任务。它能按照次序运行测试，跳过不必要的那些测试（比如你已经知道无法ping通邮件服务器的时候，应该就没有必要尝试连接它的SMTP端口），并且产生容易阅读的输出。

使用Test::More非常容易。第一步是告诉它要运行的测试的数量：


```
use Test::More tests => 5;
```

如果你不知道某个脚本要运行的测试的数量（没准你正在开发中），那么可以这样声明：

```
use Test::More 'no_plan';
```

然后我们可以书写测试的内容：

```
is(check_dns('my_server'),$known_ip,
   'DNS query returns right address for server');
is(sha2_page('http://www.example.com'),
   '6df23dc03f9b54cc38a0fc1483df6e21',
   'Home page has correct data');
```

`Test::More`只定义了包括`is()`在内的几个简单测试例程，而`check_dns()`、`sha2_page()`、`router_interface_up()`、`correct_ports_open()`或者其他需要的子例程则需要自行开发。网站工作正常时这个脚本可以产生下面这样的输出：

```
ok 1 - DNS query returns right address for server
ok 2 - Home page has correct data
```

但是，如果某个问题出现的时候，我们会看到这样的输出：

```
not ok 1 - DNS query returns right address for server
#   Failed test (test.pl at line 5)
#       got: '192.168.0.4'
#   expected: '192.168.0.6'
ok 2 - Home page has correct data
```

一旦产生这样的输出，我们可以把它传给显示问题的脚本（可能是用我们在这一章前面的“文本展现工具”一节或者“图形展现工具”一节中介绍的工具）。

因为编写测试脚本非常容易，所以可能你很快积累了一大堆的`Test::More`脚本，分别用来完成系统不同部分的测试。到了一定时候，维护这些测试脚本开始变得非常麻烦。这时候你可以使用Perl自带的`Test::Harness`模块来管理它们。使用时，只要把需要运行的文件列表传给它，它就能运行所有的测试脚本，并且返回所有成功和失败测试的报告。另外还有一些更加复杂的测试模块可以尝试。你可以参考Ian Langworth和chromatic写的《Perl Testing: A Developer's Notebook》一书（O'Reilly出版）了解更多信息。

一旦你已经完成了框架的开发（不管它是一个`Test::Havness`脚本还是一个`Test::More`子例程调用的集合），那么就有必要从`cron`、`launchd`或者`scheduler`服务定时启动它们。如果你连脚本的调度也希望使用Perl来完成，那么可以使用Roland Huß写的`Schedule::Cron`模块，它给Perl子例程提供了类似于`cron`的调度机制。

如果你需要的是更加复杂的系统监控框架，那么还可以考虑如下两个方向：

- 对于那些更加复杂，但是仍然属于Perl范围的任务来说，最好考虑使用Perl对象环境(Perl Object Environment, POE)。比如，如果你需要同时从某个大型网络的多个网络设备采集数据，并且还要同步完成数据的统计，那么POE会是非常自然的选择。POE其实是一个迷你的操作系统，其中的进程可以在内核的调度下分时同步执行。POE的学习有些难度，因为它需要OOP方面的基础知识，更重要的是还要熟悉POE特有的词汇（会话、轮子、处理程序、驱动程序等等），当然也得对多任务编程有些概念。这里我们不打算介绍POE的基础概念，因为那需要一章的篇幅，感兴趣的读者请参考Simon Cozen的《Advanced Perl Programming, Second Edition》一书（O'Reilly出版）。POE的主页也提供了不少有益的范例程序。
- 如果完全使用Perl来实现这个想法开始时有些难度，你还可以考虑对现有的系统监控软件包进行扩展。所有好的软件包都能通过定制的插件模块来进行扩展。下一节我们就会介绍插件的开发理念和典型的可用软件包。

对现有监控软件包进行扩展

现今绝大多数可扩展的监控软件包都有相似的扩展机制：它们会主动调用你开发的代码，并且希望你的程序返回某种预先定义的输出。比如每5分钟，监控系统就会调用你的程序（可能是一个Perl脚本），然后期望你的程序返回“OK”或者“NOT OK”这样的字符串。你的程序做什么检查则完全是你的自由。它可以通过SNMP查询路由器的数据包流量，并且和上次查询到的流量比较，确保它的增长合乎常见的规律；它也可以连接LDAP服务器来查看是否存在某个测试条目；或者它还可以通过某个私有API连接公司的Web 应用，检查它的运行状况。这一切都是你自行决定的，关键是对于预定的输入，你的程序可以产生正确的输出。

为了让概念更加清晰，我们可以看看具体的例子。有个叫做Big Brother (<http://www.bb4.org>和<http://www.quest.com/bigbrother/>)的监控软件包允许你自由开发插件（被称为“外部脚本”），用来向服务器汇报服务的运行状况。虽然你可以通过Perl的socket支持来连接服务器完成汇报，但最常见的办法还是通过调用bb命令来完成，注意它的参数：

```
# bb machine_name
    color_code_for_display
    status_message
system("bb mymachine green everything_groovy")
```

如果你不想自行拼接这样的命令串，还有一个第三方的模块可以帮你完成。它的名字叫做BigBrother.pm，可以在<http://www.deadcat.net>下载，它使得Big Brother的Perl插件编写变得更加容易。

基本上这方面所有软件包的扩展机制都类似于这个例子。下面让我们看看三个开源软件包在监控工具中的实现。

注意：关于选择监控软件包的两点建议：

1. 开源软件包中并不缺乏好的系统监控工具。这里介绍的软件目前都还能保持更新，也有活跃的用户社区。而其他的那些软件，比如Spong和Big Sister，虽然值得一看，但是它们的开发处于停滞状态。在挑选软件包的时候请特别注意它们将来的发展趋势。
 2. 这里只介绍开源软件包，而非任何的商业软件，这是因为它们的起点比较低，实施起来不需要大量的资金。这也是我们在插件开发方面的一个主要考虑因素。
-

Xymon

Xymon (<https://sf.net/projects/xymon/>)，曾经命名为“Hobbit” (<https://sf.net/projects/hobbitmon/>)，它是Big Brother的衍生软件包，能很好地兼容前者。所以它也支持前面例子中的bb命令。

Mon

Mon (<http://www.kernel.org/software/mon/>) 的介绍说它基本上是一个“外部监控程序的调度器，并且在监控失败的时候调用相应的告警程序”。监控程序分别针对不同的主题，而且都是通过程序或脚本来实现，所以扩展起来非常容易。软件包的配置文件`mon.cf`列出了需要运行的程序和相应的参数。程序是通过退出时返回0来表示测试通过，而这和标准的shell成功返回码是一致的。其他的返回值表明系统某处出了问题。程序还可以在出问题的时候输出详细的信息，而第一行输出会被软件包理解为总结信息。

这个程序应该很容易编写，这正是为什么Mon已经积累了很多人编写的各种服务和设备的监控程序。不过这个软件包还能进一步进行扩展。在每次启动监控程序之前，Mon会通过环境变量来告知程序上次运行的结果（比如MON_LAST_SUCCESS用来声明上次成功运行的时间，而MON_LAST_OUTPUT则含有上次脚本的输出）。Perl脚本很容易获取这些数据：

```
my $lastfailure = $ENV{MON_LAST_FAILURE};
```

有了这些数据，处理复杂情况会变得更加容易，而且也能据此决定对当前运行需要执行的测试。

Nagios

Nagios是这里介绍的软件包中最复杂的。不过好在Nagios和Mon的插件有相同的返回码机制，它也是依赖于程序的返回码以及首行输出来提供更多信息。Nagios的插件文档中

列出了可用的返回码，这也是唯一的带有插件开发文档的软件包。

Nagios对Perl插件的开发有特殊的要求，主要是因为它特殊的嵌入式Perl执行解释器，不过这些约束对其他的软件包来说也是合理的建议。比如，插件的作者有义务保证插件支持超时退出机制。这能避免因为服务器的挂起而导致插件挂起整个监控程序。所以哪怕你不打算使用Nagios，也可以试着阅读一下它的插件开发文档。

现在还剩什么？

现在我们有探测、显示和框架组件，所以对于简单的网络监控系统来说大体上架构已经完整了。最后一个要补充的组件就是分析和通知机制，而它可简单也可复杂。这里说的也就是那些分析框架采集的数据结果并决定要怎么告知相关人员的代码。我们早在第8章和第10章就介绍了这个主题，所以你应该参考这两章来补齐这个监控系统剩下的最后缺口。

本章所用模块

模块名	CPAN ID/URL	版本
Net::Ping	Bundled with Perl	
Net::Ping::External	COLINM	0.11
Win32::PingICMP	TEVERETT	0.02
Net::Netmask	MUIR	1.9012
NetPacket	ATRAK	0.04
Net::Packet	GOMOR	2.04
Net::Arping	RIIKI	0.02
SNMP::Info	MAXB	0.9.0
Nmap::Scanner	MAXSHUBE	0.8.0
Text::Autoformat	DCONWAY	1.14.0
Array::PrintCols	AKSTE	2.1
Text::FormatTable	DSCHWEI	1.01
Text::BarGraph	KBAUCOM	1.0
GD::Graph	MVERB	1.43
GD::Graph3d	WADG	0.63
GraphViz	LBROCARD	2.02
RRDs	Bundled with RRDtool	

模块名	CPAN ID/URL	版本
Test::More	Bundled with Perl	
Test::Harness	Bundled with Perl	

更多参考资料

<http://www.packetfactory.net/projects/nemesis/>是nemesis的主页。

《Silence on the Wire: A Field Guide to Passive Reconnaissance and Indirect Attacks》，由Michal Zalewski所著（No Starch Press出版），是一本关于不使用直接探测手段来查找网络及其中主机信息的书。

<http://www.tcpdump.org>是tcpdump和libpcap库的主页。

<http://www.winpcap.org>是libpcap库的Windows移植版的主页。

<http://www.insecure.org/nmap/index.html>是Nmap安全扫描器的主页。

<http://rrdtool.org>是到RRDtool的主页的指针。

<http://poe.perl.org>是POE的主页。

Randal Schwartz 在2003年11月及12月所写的关于使用Test::More模块检测站点健康状况的专栏文章可以在位于<http://www.linux-mag.com/magazine/backissues/>的《Linux Magazine》杂志的存档中找到。

《Perl Testing: A Developer's Notebook》，由Ian Langworth和chromatic所著（O'Reilly出版），是本关于Perl测试模块和选项的实用指南。

Edward Tufte编写并（在本书写作之时）自己出版了四本极好的关于信息呈现的书：《The Visual Display of Quantitative Information》《Envisioning Information》

《Visual Explanations: Images and Quantities》《Evidence and Narrative》和《Beautiful Evidence》。对于需要获取数据、加以理解并展现给其他人的开发者来说，这些书都非常值得一读。

第14章

实验性学习

抱歉在这里用了这样一个奇怪的标题。我只是不希望你的老板看到你在阅读写着“系统管理员其实只是需要更多乐趣”这样的书，而那个题目其实更适合本章。如果你就是老板（而且并非从系统管理员起家的），那么请不要告诉高层管理人员。如果你就是最高层的管理者，那么请随意转述这句话，因为没有人会信以为真的。

这个秘密其实早在我给Thomas Limoncelli的《Time Management for System Administrators》（O'Reilly出版）一书的序中就已经声明了：

其实能驱动系统管理员工作的往往就是乐趣。对他们来说，这些修补、集成、安装、构造、升级还有对系统的把玩等等全都是有趣的。所以实际上是乐趣使他们整天工作，而且回家之后还能继续乐此不疲。

在乘车的时候，我曾经听到一个大厨告诉我她在下班之后从来不会做饭。“邮差下班回家之后不愿意多走一步路”，这是她口中所说的。而我遇到的大多数系统管理员从来没有说过类似的话。你会发现他们回家之后立刻打开笔记本电脑开始做些自得其乐的事情。我自己也是这样，我太太会很乐意证明这点。对他们来说娱乐和工作的关系就像波粒二象性那样微妙。

在系统管理领域能让我尊敬的人，他们往往总是能从自得其乐的实验中学到很多东西。他们也很容易把实验中学到的东西用到工作中去。他们之所以有那么高的工作效率，就是因为他们总是知道许多问题的正确解决方法，当然也包括错误的方法。

这一章展示了如何从娱乐中积累工作的经验。虽然我并不会替你向老板争取更多的“工作娱乐时间”^[注1]，但是这一章应该能给你带来一些对乐趣的向往。

注1： Google20%的时间，有人向往吗？

漫步时间线

在2008年1月，下面的消息被发布到SAGE邮件列表（经过允许对内容进行了节选，以保护发帖人隐私）：

From: ...

Date: January 9, 2008 2:10:14 PM EST

Subject: Re: [SAGE] crontabs vs /etc/cron.[daily,hourly,*] vs.

/etc/cron.d/

出于好奇，请问有没有人知道哪种工具能把crontab^[注2]转化成甘特图？不一定要最佳实践，只要是可行方案即可。

我经常希望有什么方法能对几台机器上的crontab任务进行可视化展示。每个任务应该有相应的估计执行时间（3分钟或者3小时）。

JM

这看上去是一个有趣的系统管理相关的可视化项目，所以我决定花点时间看看它有多复杂。我会分步介绍它的实现过程。大体上我们分三步来完成这个挑战：解析crontab文件、显示时间线、产生XML输出用于绘制时间线。最后我们把它们整合在一起，从而显示出结果。

任务一：解析crontab文件

这个项目的第一个小任务是如何分析并解读标准的crontab文件。读取文件并获取它的各个字段是很容易的，不过对于我们的需求来说并无帮助，因为我们需要的是对cron的解读。我们需要某种方法来确定某段时间范围中cron中某一行会得到运行的所有时刻。

比如，我们现在有一个这样的非常基础的crontab文件^[注3]：

```
45 * * * * /priv/adm/cron/hourly
15 3 * * * /priv/adm/cron/daily
15 5 * * 0 /priv/adm/cron/weekly
15 6 1 * * /priv/adm/cron/monthly
```

注2： crontab文件是Unix核心的任务定时调度机制。请运行man 5 crontab或者man crontab以了解更多的信息。

注3： 这里我们感兴趣的主要是标准的crontab格式。我们使用的模块支持的就是这种格式，而不是扩展的格式。如果我们在命令之前加入用户名（有的场合会使用这种格式），那么用户名会被理解成命令行。

第一行说明每到整点之后的45分钟，`/priv/adm/cron/hourly`程序就会运行，所以我们会在凌晨1:45、2:45、3:45等时间运行这个程序。第二行说明每天凌晨的3:45运行`/priv/adm/cron/daily`，剩余几行也大致相仿。

要分析出具体的运行时间不是不可以，只是非常痛苦。还好我们手头还有Piers Kent写的`Schedule::Cron::Events`模块，节省了我们很多时间。它会通过调用Abhijit Menon-Sen的`Set::Crontab`模块来分析`crontab`文件的每一行，然后通过简单的接口返回我们需要的具体运行时刻。

要使用`Schedule::Cron::Events`，我们只需要给它传递两个信息：`crontab`中我们感兴趣的那一行以及我们希望它计算的时间起点：

```
my $event = Schedule::Cron::Events( $cronline, Seconds => {some time} );
```

（这里的`{ some time }`是用纪元以来的秒数表示的）。

一旦创建了对象，只要不断对它调用`$event->nextEvent()`就能返回描述下次运行时间的字段（年、月、日、时、分、秒）了。

任务二：显示时间线

创建漂亮的时间线可不是一个平常的任务，所以我们应该考虑采用其他人的解决方案。虽然Perl也有不少成熟的时间线展示（比如`Data::Timeline`）和显示（比如`Graph::Timeline`）模块，但是有一种创建时间线的方法特别棒，所以我决定舍弃Perl。MIT的SIMILE项目创建了一个名叫`Timeline`的工具，用来通过DHTML基于的AJAX widget来可视化基于时间的事件。你可以从<http://simile.mit.edu/timeline/>看到更多相关信息。

要使用这个widget，我们需要创建两个文件：HTML文件用于引用、初始化并显示MIT的widget，另外还需要一个XML文件用于存放我们需要显示的时间。而后面这个文件是下一节的任务，目前我们先展示这个HTML文件。不过我得向你坦白，我的JavaScript技巧还很稚嫩，所以我主要是照抄SIMILE项目的指南。如果这个语言对你来说也有点偏门，那么请留意其中的注释（用`<!-- -->` 和`//`标记的内容）。

```
<!DOCTYPE html PUBLIC "-//W3C//DTD HTML 4.01//EN">
<html>
  <head>
    <!-- 该 widget 的 源码地址 -->
    <script src="http://simile.mit.edu/timeline/api/timeline-api.js"
      type="text/JavaScript">
    </script>

    <script type="text/JavaScript">
```



```

function onLoad() {
    // 变量 tl 将持有要创建的时间线对象
    var tl;
    // 准备好从哪个事件来源对象获取时间线数据
    var eventSource = new Timeline.DefaultEventSource();

    // 创建两个水平时间轴，一个用于显示小时，另一个用于显示对应的天。
    // 注意：两者都设为使用我所在的时区（即 -5 GMT）
    var bandInfos = [
        Timeline.createBandInfo({
            eventSource: eventSource,
            timeZone:    ?5, // 我的时区在波士顿
            width:       "70%",
            intervalUnit: Timeline.DateTime.HOUR,
            intervalPixels: 100 }),
        Timeline.createBandInfo({
            timeZone:    ?5,
            width:       "30%",
            intervalUnit: Timeline.DateTime.DAY,
            intervalPixels: 100 }),
    ];

    // 保持两根时间轴同步，并设置为高亮显示连接
    bandInfos[1].syncWith = 0;
    bandInfos[1].highlight = true;

    // 创建时间线对象并从 output.xml 文件加载数据
    tl = Timeline.create(document.getElementById("cron-timeline"), bandInfos);
    Timeline.loadXML("output.xml", function(xml, url) {
        eventSource.loadXML(xml, url); });
    }

    // 此处借用教程中给出的示范代码
    var resizeTimerID = null;
    function onResize() {
        if (resizeTimerID == null) {
            resizeTimerID = window.setTimeout(function() {
                resizeTimerID = null;
                tl.layout();
            }, 500);
        }
    }
}
</script>
<title>My Test Cron Timeline</title>
</head>

```

<!-- 在当前页面加载完成后运行我们定制的初始化代码；同样，在调整浏览器窗口大小后也调用定制的代码 -->

```
<body onload="onLoad();" onresize="onResize();">
```

<!-- 在此处显示实际的时间线 -->

```
<div id="cron-timeline"
    style="height: 150px;
```

```

        border: 1px solid #aaa">
    </div>

</body>
</html>

```

这里我不会花力气向你解释这个文件的意思，请自行参考Timeline tutorial(<http://simile.mit.edu/timeline/docs/create-timelines.html>)。

最后还要展示的是完成这个任务所需的范例XML文件，它的文件名是`output.xml`。这将让你对widget期望我们提供什么数据有所了解。下面显示的是2008年1月的cron事件：

```

<data>
  <event start="Jan 01 2008 00:45:00 EST" title="/priv/adm/cron/hourly"></event>
  <event start="Jan 01 2008 01:45:00 EST" title="/priv/adm/cron/hourly"></event>
  <event start="Jan 01 2008 02:45:00 EST" title="/priv/adm/cron/hourly"></event>
  <event start="Jan 01 2008 03:45:00 EST" title="/priv/adm/cron/hourly"></event>
  ...
  <event start="Jan 01 2008 03:15:00 EST" title="/priv/adm/cron/daily"></event>
  <event start="Jan 02 2008 03:15:00 EST" title="/priv/adm/cron/daily"></event>
  <event start="Jan 03 2008 03:15:00 EST" title="/priv/adm/cron/daily"></event>
  <event start="Jan 04 2008 03:15:00 EST" title="/priv/adm/cron/daily"></event>
  ...
  <event start="Jan 06 2008 05:15:00 EST" title="/priv/adm/cron/weekly"></event>
  <event start="Jan 13 2008 05:15:00 EST" title="/priv/adm/cron/weekly"></event>
  <event start="Jan 20 2008 05:15:00 EST" title="/priv/adm/cron/weekly"></event>
  <event start="Jan 27 2008 05:15:00 EST" title="/priv/adm/cron/weekly"></event>
  <event start="Jan 01 2008 06:15:00 EST" title="/priv/adm/cron/monthly"></event>
</data>

```

任务三：输出正确的XML文件

目前为止我们已经解决了这个项目主要的技术难题，弄清楚了需要的数据以及如何使用这些数据。最后一个任务就是确保数据的格式化输出正确无误。这里我们需要创建一个有特定标签和内容的XML文件。如同在第6里看到的，使用Perl创建XML文件的方法非常多。我们会使用那里介绍的Joseph Walton维护的XML::Writer模块来完成任务。这需要我们编写下面的代码：

```

use IO::File;
use XML::Writer;

# 设定输出资源
my $output = new IO::File('>output.xml');

# 创建一个新的 XML::Writer 对象，并打开一些选项让输出更好看
my $writer
    = new XML::Writer( OUTPUT => $output, DATA_MODE => 1, DATA_INDENT => 2 );

# 创建一个 <sometag> 起始标签，并指定相关属性的取值
$writer->startTag('sometag', Attribute1 => 'value', Attribute2 => 'value' );

```

```
# 顺便提一下：这里不指定标签名也没问题，该方法会自动使用最近打开的标签名
$writer->endTag('sometag');

$writer->end();
$output->close();
```

思路汇总

现在我们有所有需要的技术组件，让我们看看最终的脚本吧。这里我只会解释那些没有介绍过的新代码。

首先，我们载入所有需要的模块：

```
use Schedule::Cron::Events;
use File::Slurp qw( slurp ); # 用该模块的 slurp 函数一次读入 crontab 文件的内容
use Time::Local;             # 用于日期格式惯例
use POSIX qw(strftime);      # 用于日期格式化
use XML::Writer;
use IO::File;
```

现在我们去采集Schedule::Cron::Events模块需要的信息。这里我们需要告诉Schedule::Cron::Events从何时开始罗列事件（也就是说从哪天开始）。从某种角度来看，显示一个展示本月事件的时间线可能是有用的，所以我们需要获得本月第一天的开始自纪元起的秒数（作为另一个参数）：

```
my ( $currentmonth, $currentyear ) = ( localtime( time() ) )[4,5];
my $monthstart = timelocal( 0, 0, 0, 1, $currentmonth, $currentyear );
```

另外，我们还需要把crontab文件读入内存并且开始准备XML输出文件：

```
my @cronlines = slurp('crontab');
chomp(@cronlines);

my $output = new IO::File('>output.xml');
my $writer
    = new XML::Writer( OUTPUT => $output, DATA_MODE => 1,
                      DATA_INDENT => 2 );

$writer->startTag('data');
```

下面我们需要开始遍历crontab文件。在遍历过程中，我们要对找到的每一行罗列出它的所有启动时间。Schedule::Cron::Events提供的nextEvent()能无限地推导出所有启动时间，所以我们必须判断何时终止。既然刚才我们决定要把时间线的显示限制在本月，那么一旦nextEvent()返回的日期不在本月范围就可以停止。所以下面的代码基本上就是这样做的，把crontab中除了注释以外的行传递给Schedule::Cron::Events，另外还带上本月开始的启动时间作为第二个参数。然后不断产生下一个启动时间，直到它超出本月范围：

```
foreach my $cronline (@cronlines) {
    next if $cronline =~ /^#/;          # 跳过注释行
    next if $cronline =~ /^#\s*\w\s*=/; # 跳过变量定义行
    my $event
        = new Schedule::Cron::Events( $cronline, Seconds => $monthstart );

    my @nextevent;
    while (1) {
        @nextevent = $event->nextEvent;

        # 如果日期超过当前月份就结束本次循环
        last if $nextevent[4] != $currentmonth;
    }
}
```

对于每个事件，我们需要生成一个<event></event>元素，其中有显示事件时间的启动属性和列出要在那个时间运行的*cron*命令的标题属性。我们会使用来自POSIX模块的*strftime()*函数来产生Timeline widget需要的时间格式。在完成所有任务之后，我们可以关闭XML文件的闭合标签，然后停止XML::Writer的处理，最终关闭文件：

```
$writer->startTag('event',
    'start' => strftime('%b %d %Y %T %Z',@nextevent),
    'title' => $event->commandLine(),
);
$writer->endTag('event');
}

$writer->endTag('data');
$writer->end();
$output->close();
```

如果我们知道每个事件的持续时间，还可以给每个元素加上一个<end></end>属性，但是要知道，*cron*任务持续时间的信息可不容易获得，哪怕估算起来也不简单（请回想本节开头的邮件中的需求）。不过这里可以假定：我们已经编写了许多其他代码，用来分析*crontab*文件中每一行程序的日志，从而获取这个信息。

那么，这段代码工作起来如何？图14-1展示的是这个widget载入新数据之后的一个截屏。

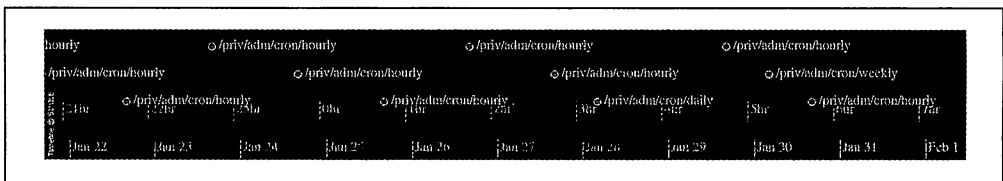


图14-1：从简单的*crontab*产生的时间线

这个输出我自己很满意，因为现在我可以一个月的范围内前后滚动时间窗口。

不过我还是意识到这段代码并没有完全实现原本的要求，原因有二：

1. 这并不是甘特图，因为那需要对*cron*任务进行分析并找出它们之间的关联。
2. 这个图还不能显现多台主机的时间线。

第一个缺陷看起来很难补足。如同另外一个人对这封邮件的答复所指出的那样，在这种需求中依赖性的跟踪实在是太复杂了，而这不可能在这一章介绍。而第二个缺陷则非常容易补足，只需要我们打开更多的*crontab*文件并且重复分析过程就可以了。

尽管代码还有这些缺陷，但是产生的效果已经让我非常满意了。于是我回复那封邮件的发件人，请他给我一些测试数据，以便让我看到代码在实际环境中的运行效果。而他也非常好心地给我发回了一些工作环境中的*crontab*文件。于是我启动自己的脚本（当然还修改了一点HTML文件，以增大显示区域来适应更多的数据），产生了如图14-2所示的输出。收到我发给他的图片之后，他告诉我这个效果非常酷。

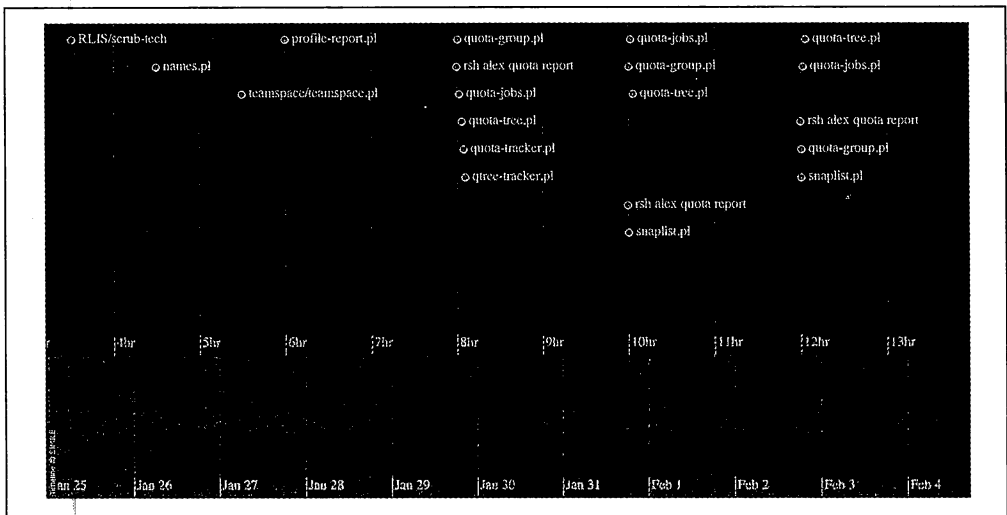


图14-2：实际环境中*crontab*的时间线

总结：我们可以从中学到什么？

为了完成这个项目，我们需要学习的东西：

- 如何与*crontab*文件打交道
- 如何使用SIMILE的Timeline widget（一旦掌握它，下次遇到这种事件时间线可视化的任务就方便了。你可以用它来证明设备的失效，或者用来进行项目的计划）
- 一些JavaScript

(更加不用说的是分解问题以便逐个解决的技巧。)

地理编码的乐趣

维基百科对地理编码的定义是“从街道地址、邮编等其地理数据找出关联的地理坐标（如经纬度）的过程。”这个词还可以指尝试按地理定位某个IP地址的过程。正是因为能把虚拟和现实世界联系起来，地理编码变得非常有趣。

邮政地址的地理编码

让我们从一个标准的地理编码任务开始：如果给定一个邮政地址，是否可以在地球上定位这个地址，这样我们可以在地图上绘出来呢？虽然商业软件很容易就能做到，但是其实自己动手实现并不容易，主要的问题是数据的准确性值得推敲。邮政地址往往并不那么精确，地理数据可能有误，另外还要面对人力和自然力量对环境的改变。

注意：第一个声明：我和这些服务的供应商并没有商业上或其他方面的联系，只是偶尔会支付一些并不高昂的使用费，以便能持续使用他们的服务。

第二个声明：当住在美国的人谈到地理编码的时候，他们其实往往指的是“北美地理编码”，而且并不是很在乎它是否可以在美国以外的地方使用。除了所谓的“美国中心”主义之外，这也和美国政府免费提供高质量的地理信息有关，这在其他国家往往不能做到。如果你感兴趣的是美国以外的地理编码，那么可以试试NAC Geographic Products, Inc. (<http://www.nacgeo.com>)的商业化服务（在低预算的前提下），它可能会提供你需要的信息。

如果我们不打算使用那些昂贵的地理编码产品，还是有一些可选的免费服务的。Perl爱好者最希望的服务应该是`geocoder.us`，它不但提供免费的网络服务，还在CPAN上发布了`Geo::Coder::US`模块（它甚至还能帮助你设立自己的网络服务）。`geocoder.us`能提供不同风格的网络服务，包括XML-RPC、SOAP、REST以及“纯文本”REST。我们打算使用XML-RPC作为例子，因为相应的代码很简单：

```
use XMLRPC::Lite;

my $reply = XMLRPC::Lite
    -> proxy ( 'http://rpc.geocoder.us/service/xmlrpc' )
    -> geocode('1005 Gravenstein Highway North, Sebastopol, CA')
    -> result;

foreach my $answer (@{$reply}){
    print 'lat: ' . $answer->{'lat'}
        . ' long: ' . $answer->{'long'} . "\n";
}
```

首先我们载入XMLRPC::Lite模块（它是SOAP::Lite发行包的内嵌模块）。它的proxy()方法用来指定查询的站点，而不是顾名思义的Web代理（或者其他的什么代理）。我们通过geocode()方法进行远程调用，并要XMLRPC::Lite返回数据。

表面上看起来打印结果的代码应该不太复杂。不过要注意geocode()返回的是哈希的列表，每个哈希都是一个返回的查询结果。有些查询可能会导致多个结果（比如你查询“300 Park, New York, NY”，那么可能意味着300 Park大道，也可能代表300 Park路，或者300 Park街）。在塞瓦斯托波尔只有一个Gravenstein Highway North，所以下面这个简单的代码就能处理所有结果（不过这样做不太严谨）：

```
print 'lat: ' . $reply->[0]->{'lat'} .  
      'long: ' . $reply->[0]->{'long'} . "\n";
```

如果你不喜欢geocoder.us返回的查询结果，还有一些便宜的地理编码服务可供选择，比如雅虎以REST方式提供的服务（每天支持5000次免费查询）。为了使用雅虎的服务，我们需要从<http://developer.yahoo.com/wsregapp/>获取免费的应用ID。有了这个ID，我们就可以使用<http://developer.yahoo.com/maps/rest/V1/geocode.html>描述的API服务了。下面就是具体的代码：

```
use LWP::Simple;  
use URI::Escape;  
use XML::Simple;  
  
# 用法：脚本名 <要取得 geocode 的地理位置>  
  
my $appid = '{your API key here}';  
my $requrl = 'http://api.local.yahoo.com/MapsService/V1/geocode';  
  
my $request  
    = $requrl . "?appid=$appid&output=xml&location=" . uri_escape( $ARGV[0] );  
  
my $response = XMLin( get($request), forcearray => ['Result'] );  
  
foreach my $answer ( @{ $response->{'Result'} } ) {  
    print "Lat: $answer->{Latitude} Long: $answer->{Longitude} \n";  
}
```

使用REST接口的好处在于它们查询起来非常简单。只要你知道如何在Perl中使用GET或者PUT请求来获取网页，就可以使用REST接口。在上面的例子中，我们构造了一个URL，它其实就是雅虎的REST请求URL再加上一些参数。这里我们需要加上appid、期待的输出格式以及URL编码的查询地址。最后这个URL被传递给LWP::Simple的get()例程，于是我们可以通过XML::Simple来解析输出。

如果地理编码服务器返回的是单个响应，XML::Simple则会返回单个哈希。如果返回的是多个响应（还记得前面提到过的地址二义性吗？）那么它会返回一个包含哈希列表的

哈希，而列表中的每个哈希都是一个答案。在分析并显示结果的时候，我们也可以通过`ref()`调用来区分两种不同的哈希，不过那实在有些麻烦。所以我们给`XML::Simple`加上了`forcearray=>['Result']`参数，强制它总是以哈希列表的哈希的格式输出，如同在第6章实现的那样。最后，我们使用`foreach`来遍历列表并输出结果。

注意：如果这段代码对你来说还是有些麻烦，那么还可以使用Ask Bjørn Hansen写的`Geo::Coder::Yahoo`模块。这个模块提供了两个函数，一个用来创建搜索对象，另一个用来调用地理编码API。API调用的返回结果是哈希列表的格式，所以不必再做XML的解析。现在你可以自己决定使用哪个方法。

现在我们已经展示了一些把地址转换成对应经纬度的方法，那么该如何使用这些信息呢？最典型的应用就是在地图上画出这个信息点位。支持这个功能的网络服务很多，包括Google Maps、Yahoo! Maps和TerraServer。另外，要弄得更加炫目，你还可以生成KML或者压缩过的KMZ文件来传给Google Earth，然后可以在点位之间进行飞览。

大多数时候，把点位画到地图上都需要我们处理HTML和复杂的JavaScript。不过对于Perl爱好者来说，我们是非常幸运的，因为Nate Mueller已经把这些集成在`HTML::GoogleMaps`模块中了。这个模块使用起来非常简单，下面的CGI脚本就能在地图上标出O'Reilly总部的位置：

```
use HTML::GoogleMaps;

# '1005 Gravenstein Highway North, Sebastopol, CA'
# 当然我们也可以指定具体地址，然后交给 Geo::Coder::Google 模块调用取得经纬度
my $coords = [ ?122.841571, 38.411239 ];

my $map
    = HTML::GoogleMaps->new( key => '{your API KEY HERE}' );
$map->center($coords);    # 将该地址作为地图中心显示
$map->v2_zoom(15);        # 比默认缩放级别更近一点

# 在该地址上添加一个点标，并用给定的 HTML片段作为标签
# （不要修改标签的大小）
$map->add_marker(
    point    => $coords,
    noformat => 1,
    html     => "<a href='http://www.oreilly.com'>O'Reilly</a> HQ"
);

# 添加一些地图控制组件（缩放杆等等）
$map->controls( 'large_map_control', 'map_type_control' );

# 创建显示地图所需的实际 HTML/JavaScript 代码
my ( $head, $map_div ) = $map->onload_render;

# 输出为 HTML （加上 CGI 脚本所需要的 Content-Type 头）
```



```

print "Content-Type: text/html\n\n";
print <<"EOH";
<html>
  <head>
    <title>Otter Demo</title>
    $head
  </head>
EOH

print
  "<body onload=\"html_googlemaps_initialize()\" onunload=\"GUnload()\">
    $map_div </body> </html>\n";

```

产生的输出效果如图14-3所示。

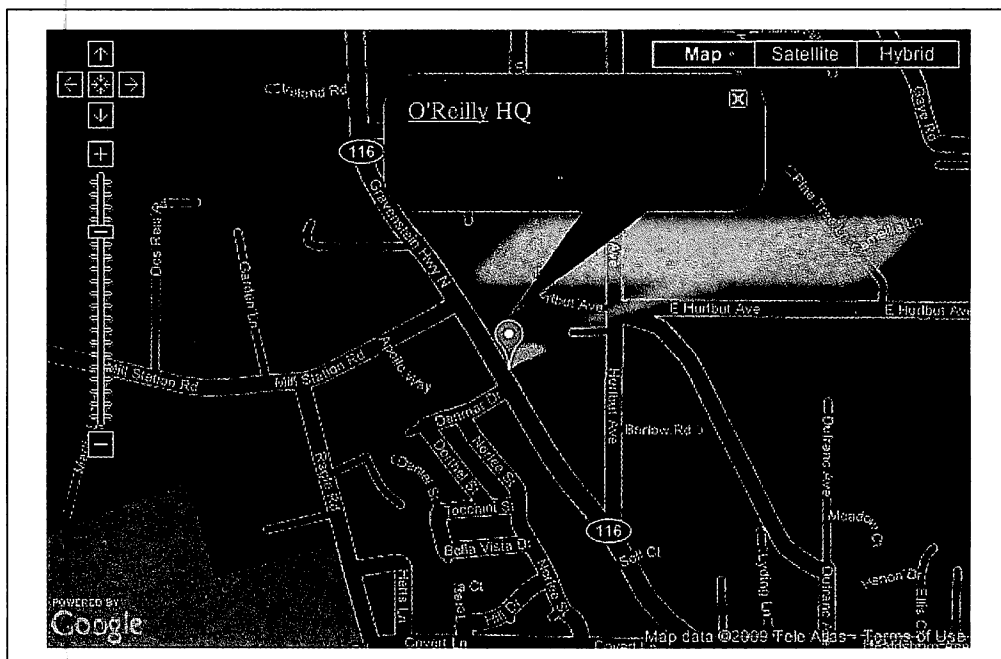


图14-3: HTML::GoogleMaps产生的地图样例

如同雅虎的服务一样，使用谷歌地图之前需要从<http://code.google.com/apis/maps/signup.html>获取API key。

不论是谷歌地图还是其他的服务都有更多值得探索的功能，请查阅相关服务和产品的文档了解细节。

IP地址的地理编码

现在看来从邮政地址定位某个地理坐标是非常可靠的事情，哪怕在中西部你也可以轻松

定位某个神秘的办公室。不过如果你能根据网址来定位某个人的位置就显得更加神秘了，因为这个转换有效地突破了虚拟和现实世界的边界。

第一步是把域名转化为IP地址。这非常简单，只要使用本书前面介绍过的`Net::DNS`模块就可以了：

```
use Net::DNS;

my $resolv = Net::DNS::Resolver->new;

my $query = $resolv->search( $ARGV[0] );

die 'No response for that query' if !defined $query;

# 仅打印找到 A 资源记录的 IP 地址
foreach my $resrec ( $query->answer ){
    print $resrec->address . "\n" if ($resrec->type eq 'A');
}
```

大多数情况下你不会对那些拥有许多IP地址的域名进行地理编码，但是这段代码确实能返回某个网址对应的所有IP地址。另外请注意，如果你要经常进行这样的域名解析（比如从某个日志文件中找到的域名），请尽可能对查询结果进行缓存，类似在第11章中做的那样。另外如果地址数量庞大，你可能需要使用`adns` (<http://www.chiark.greenend.org.uk/~ian/adns>) 这样的异步DNS库处理平行查询，可以使用`Net::ADNS`或者`EV::ADNS`模块来调用`adns`。

现在我们已经有了IP地址，可以从另外一个网络服务来获取所需要的信息。很多非常廉价的服务供应商都能支持海量查询。下面的例子使用的是MaxMind的服务，因为这个我最熟悉^[注4]。你可以从<http://www.maxmind.com/app/ip-location>了解这个服务的相关信息。

MaxMind和其他一些供应商一样，都提供网络服务接口和预付费的数据库下载两种查询方式，而后者的查询速度更快。我们会展示两种查询方法，因为相应的代码都很简单。

对于MaxMind的网络服务来说，我们还需要构造一个简单的HTTP GET（或者PUT，随你喜欢）请求，类似这一章前面展示的雅虎API。主要的区别在于输出格式，这里我们获得的是逗号分隔的字符串（CSV）输出，而不是XML格式的输出：

```
use LWP::Simple;
use Text::CSV_XS;    # 这是比 Text::CSV 更快的版本

# 用法：脚本名 <要取得 geocode 的 IP 地址>
```

注4： 一个小道消息：MaxMind公司的创办者就是著名的Perl高手T. J. Mather (<http://search.cpan.org/~tjmather>)。

```

my $maxmkey = '{your API key here}';

my $requrl = "http://maxmind.com:8010/f?l=$maxmkey&i=$ARGV[0]";

my $csvp = Text::CSV_XS->new();    # (或者用 Text::CSV->new())

$csvp->parse( get($requrl) );

my ( $country, $region, $city,      $postal,
    $lat,      $lon,      $metro_code, $area_code,
    $isp,      $org,      $err
) = $csvp->fields();

```

而另一种方式是下载数据库，并且告诉MaxMind提供的模块数据库的位置。可选的数据库类型包括压缩过的二进制文件，或者用来导入至SQL数据库的CSV格式的文件。下面展示的是二进制文件的使用：

```

use Geo::IP;

my $gi = Geo::IP->open( 'GeoIPCity.dat', GEOIP_STANDARD );

my $record = $gi->record_by_name( $ARGV[0] );

print join( "\n",
    $record->country_code, $record->country_code3, $record->country_name,
    $record->region,      $record->region_name,   $record->city,
    $record->postal_code,  $record->latitude,     $record->longitude,
    $record->time_zone,    $record->area_code,    $record->continent_code,
    $record->metro_code );

```

好了，最后一个有趣的地理编码相关的项目是获得某地的天气，因为之前我们已经通过地理编码从IP地址获得了一个美国的地址，也包括邮编。从邮编获取天气状况是非常容易的。我至少知道四个免费的美国天气服务：

- NOAA提供的基于SOAP的National Weather Service（详细信息请参考<http://www.weather.gov/xml/>）。
- Weather.com提供的基于XML的服务（详细信息请参考<http://www.weather.com/services/xmlsoap.html>）。不过如果你要在你自己的网站上使用这个服务，你必须遵守很多条款。
- 雅虎以RSS的方式提供天气信息（参考<http://developer.yahoo.com/weather/>）。你只需要使用XML::RSS这样的模块就能解析RSS，有人甚至直接使用XML::Simple模块。
- <http://www.rssweather.com>也以RSS的方式提供了许多天气信息服务。

这个项目需要我们集成前面几节学到的知识。下面的CGI脚本能够从访问者的IP地址来解析出邮编，并且从雅虎查询当前的天气情况和天气预报：

```
use LWP::Simple;
use Text::CSV_XS;
use XML::RSS;

my $maxmkey = '{your API key here}';
my $requrl = "http://maxmind.com:8010/f?l=$maxmkey&i=$ENV{'REMOTE_ADDR'}";

my $csvp = Text::CSV_XS->new();

$csvp->parse( get($requrl) );
my ($country, $region, $city, $postal,
    $lat, $lon, $metro_code, $area_code,
    $isp, $org, $err
) = $csvp->fields();

print "Content-Type: text/html\n\n";
print << "EOH";
    <html><head><title>Otterbook test</title></head>
    <body>
EOH
print "<p>Hi there " . $ENV{'REMOTE_ADDR'} . "!</p>\n";

if ($postal) {
    my $rss = new XML::RSS;
    $rss->parse( get("http://xml.weather.yahoo.com/forecastrss?p=$postal") );
    print '<h1>' . $rss->{items}[0]->{'title'} . "</h1>\n";
    print $rss->{items}[0]->{'description'}, "\n";
}
print "</body></html>\n";
```

总结：我们学到了什么？

为了实现这个项目，你得学习：

- 最简单的XML-RPC调用（用于大多数网络服务）。
- 如何使用各种地理编码服务。下次你的老板问“如何了解我们网站的点击都是来自于哪些地方？”这样的问题的时候就非常好办了。另外，下次需要把用户重定向到最近的镜像站点也变得很容易。
- 如何使用雅虎和谷歌的API来获取地图、地址信息、天气预报等服务。

与MP3打交道

这些年来我一直热衷于认识更多系统管理员，所以我会参加很多像LISA（Large Installation System Administration，参考<http://www.usenix.org>获取更多详细信息）这样

的会议，我发现他们有一个共同特点，就是喜爱（甚至酷爱）音乐。其中很多人都拥有大量的MP3或者Ogg/FLAC/Shorten格式的音乐（都是从合法渠道获得的）^{注5}。

我们中很多人还对手头的音乐进行精心的维护，确保每个文件都有合适的标签（有时候只是为了让MP3播放器显示起来干净整洁）。有些人会把这些音乐文件当成一个有趣的数据库来维护。下面的这些资源我都曾经用过。

为了逐个维护文件，有两个模块我一直使用：MP3::Info，最早是Chris Nandor开发的，现在维护者是Dan Sully；另一个是Ilya Zakharevich写的MP3::Tag，这个模块在某些情况下也会使用MP3::Info模块。后面的模块在需要把信息写入MP3文件的时候功能更全，不过这里我将展示的是MP3::Info。

这个模块最关键的函数是get_mp3info()调用。只要给它一个文件名，就能获得一个包含文件信息的哈希引用。比如下面的代码：

```
use MP3::Info;

my $mp3 = get_mp3info($file);
```

就会给你一个获取文件信息的方法，如下：

```
$mp3->{SECS};    # 音频时长，单位为秒
$mp3->{BITRATE};  # 以 kbps 表示的比特率
```

另外还有一个类似的调用get_mp3tag()，更加有趣。有了它，我们可以这样：

```
my $mp3 = getmp3tag($file);
```

然后就能获得这样的数据：

```
DB<1> x $mp3

0 HASH(0x439c00)
  'ALBUM' => 'Feel Good Ghosts'
  'ARTIST' => 'Cloud Cult'
  'COMMENT' => 'ISRC US 786 08 00002'
  'GENRE' => 'Other'
  'TAGVERSION' => 'ID3v1.1 / ID3v2.3.0'
  'TITLE' => 'Everybody here is a Cloud'
  'TRACKNUM' => 2
  'YEAR' => 2008
```

或者这样的：

```
DB<1> x $mp3
```

注5：我可以很自豪地说系统管理员的收入还不错，而且他们往往对花天酒地没兴趣，所以可以买到不少的音乐。

```

0 HASH(0x95a6dc)
  'ALBUM' => 'Little Creatures'
  'ARTIST' => 'Talking Heads'
  'COMMENT' => '6F091209'
  'GENRE' => 'rock/pop'
  'TAGVERSION' => 'ID3v2.3.0'
  'TITLE' => 'Road To Nowhere'
  'TRACKNUM' => '9/9'
  'YEAR' => 1985

```

MP3::Info还能让你给文件设置标签，不过这不是我们的兴趣所在。

对于我来说，更加有趣的是在此应用File::Find::Rule系列模块。有一个叫做File::Find::Rule::MP3Info的模块能让我们写出这样的代码（摘自模块说明文档）：

```

use File::Find::Rule::MP3Info;

# 哪些 mp3 文件还没设置艺术家名称标签?
my @mp3s = find( mp3info => { ARTIST => '' }, in => '/mp3' );

# 哪些 mp3 文件是超过3分钟的?
@mp3s = File::Find::Rule::MP3Info->file()
    ->mp3info( MM => '>=3' )
    ->in( '/mp3' );

# 哪些文件是 Kristin Hersh 或者 Throwing Muses 演绎的?
# 有时候我可是个对标签内容大小写不太在意的人
@mp3s = find( mp3info =>
    { ARTIST => qr/(kristin herish|throwing muses)/i },
    in => '/mp3' );

```

我不会在这里介绍一个完整的项目，不过希望你已经可以想象出这样的脚本将来可以做的事情了。

总结：我们可以学到什么？

在这个项目中，你应该学到了：

- 如何与MP3文件打交道（不算特别完整，但是应该够用）。
- 如何更加熟练使用File::Find::Rule模块。在这个主题之外，还有File::Find::Rule::Permissions这样的模块，能让我们写出如下代码：

```

# 当前目录下有哪些文件是 'nobody' 用户可以读取的?
@readable = File::Find::Rule::Permissions->file()
    ->permissions(isReadable => 1, user => 'nobody')
    ->in('.');

```

正是这些驱使我一再介绍File::Find::Rule模块，不只是在第2章，在我的工作中我也会时常拿来使用。

临别演出

最后还要展示一个集成了网页抓取和地理编码/地图显示技巧的小项目。

要知道这本书之所以能有这么高的质量，是因为在幕后有很多的技术审阅者（可以参考本书的出版说明部分的大段感谢之词）。在设立本书审阅体系的时候，我发现那些花了大量时间审阅的人在地理位置上分布得非常分散。我的观察是基于他们的邮件地址和发信的时区。为了增加乐趣，我请求审阅者在一个wiki页面上登记他们的地址。然后我便更加确信他们分布得非常广泛，于是这个项目诞生了。这一节我们会从那个页面获取信息并在地图上显示他们的位置。

第一步：用WWW::Mechanize从Wiki页面获取数据

下载单个页面的内容非常简单，如同本章前面的LWP::Simple范例展示的那样。不过，一旦页面有保护机制就显得比较麻烦了。在这个例子中，页面内容是在受密码保护的wiki页面上的，此wiki页面是我们用于调整本书工作的trac(<http://trac.edgewall.org>)实例的一部分。为了访问那个页面，我们需要先向一个Web表单提交信息，从而完成登录。

我遇到这种网络爬虫和网页抓取等问题的时候几乎总是会使用Andy Lester写的WWW::Mechanize模块（包括那些相关模块），这是我知道的最擅长处理这类任务的工具。让我们先看看这个模块的用法，然后再解决密码保护页面的问题。

几乎所有的WWW::Mechanize脚本的开头都是这样的：

```
use WWW::Mechanize;

my $mech = WWW::Mechanize->new();

# get() 方法也可以接受 ":content_file" 这样的参数，
# 以便直接将获取的信息存入外部文件中
$mech->get($url);
```

首先我们创建一个新对象，用以获取页面。如果我们还安装了提供SSL支持的模块（Crypt::SSLeay或者IO::Socket::SSL），那么就可以对http和https页面发出get()调用。

如果我们想要获得页面内容，可以这样调用：

```
my $pagecontents = $mech->content();
```

把content()方法获取的内容传给其他模块进一步处理也很常见。下面一段我们就会这样做。

到目前为止的代码都很简单。其实对于这么简单的事情，LWP::Simple也能漂亮地完成。让我们把难度提高：

```

use WWW::Mechanize;

my $mech = WWW::Mechanize->new();

$mech->get( 'http://www.amazon.com' );
$mech->follow_link( text => 'Help' );
print $mech->uri . "\n";

# 这会输出类似这样的地址:
# http://www.amazon.com/gp/help/customer/display.html?ie=UTF8&nodeId=508510

```

看到有趣的地方了吗？这里用WWW::Mechanize取得了Amazon.com的首页，然后找到了页面上标着“Help”的链接。然后程序模拟浏览器那样点击这个链接并访问URL相应的页面。如果我们在代码的最后调用\$mech->content()，就能获得新页面的内容。

如果需要，我们还可以做得更加酷一些：

```

$mech->follow_link ( text_regex => qr/rates.*policies/ );
# 或者
$mech->follow_link ( url_regex => qr/gourmet.*food/ );

```

第一行代码能搜索并跟随其文本符合指定正则表达式的第一个链接。这使得我们不必知道非常精确的文本（这对那种动态生成的页面是非常典型的）。第二行的逻辑也大致相仿，只不过正则表达式是针对 URL 而不是链接的文本。

follow_link()还有一些其他用法。比如，可以使用url => 'http://...'选项来指示需要跟随的链接地址。另外还有一个比较容易出错的n =>选项，可以指示follow_link()去跟随页面上的第N链接。这些选项还能联合使用，也就是说如果你需要跟随的是“help”相关的第三个链接，而且地址中带有“forum”这样的字符串，就可以这样写：

```

$mech->follow_link( text => 'help', url_regex => 'forum', n => 3 );

```

如果我们只是需要找到页面上的链接，并不需要立刻访问它指向的页面，那么还可以使用WWW::Mechanize的find_link()和find_all_links()方法，它们和follow_link()有相同的选择器参数。另外WWW::Mechanize还能用来定位页面中的图片，相应的方法是find_images()和find_all_images()，参数也大致相仿。

现在让我们看看如何使用它来登录网站，以获取需要的内容。只要你能了解页面上的表单需要的信息，WWW::Mechanize就能帮你完成登录表单的提交，它自带的mech-dump工具能够帮助你理解表单的结构。在安装WWW::Mechanize的时候你可以选择一并安装mech-dump。

mech-dump使用了WWW::Mechanize模块来完成底层的工作，从而使你对WWW::Mechanize的工作对象有清晰的认识。它可以做的事情有四种：

- 显示页面上的所有表单
- 显示页面上的所有链接
- 显示页面上的所有图片
- 显示所有以上的对象

让我们看看它的工作过程：

```
$ mech-dump --links http://www.amazon.com
http://www.amazon.com/access
/
/gp/yourstore/ref=pd_irl_gw?ie=UTF8&signIn=1
/gp/yourstore/home/ref=topnav_ys_gw
...
```

我对列表进行了节选，因为：

```
# 统计该页面上的链接总数
$ mech-dump --links http://www.amazon.com|wc -l
247
```

虽然找链接也有用，不过在处理表单时这个命令才真正显出它的实力（我们马上就会用到）：

```
$ mech-dump --forms http://www.boingboing.net

GET http://www.google.com/search
ie=UTF-8 (hidden readonly)
oe=UTF-8 (hidden readonly)
domains=boingboing.net (hidden readonly)
sitesearch=boingboing.net (hidden readonly)
q= (text)
btnG=Search (submit)

POST http://www.feedburner.com/fb/a/emailverify
email= (text)
url=http://feeds.feedburner.com/~e?ffid=18399 (hidden readonly)
title=Boing Boing (hidden readonly)
loc=en_US (hidden readonly)
<NONAME>=Subscribe (submit)
```

输出显示了每个表单包含的所有字段。有些是隐藏字段，只有网页的作者才能设置，而剩下的那些浏览器可见的字段则对我们更有意义。比如Boing Boing blog是允许通过Feedburner提供的服务来进行邮件订阅的。mech-dump的输出使得我们了解需要填入的字段名叫email（而不是address或者user_email，也不是其他相似的名字）。

如果我们对wiki所在的trac站点调用mech-dump，它会输出：

```
$ mech-dump --forms http://otterbook.example.org/otterbook/wiki
```

```

GET http://otterbook.example.org/otterbook/search

POST http://otterbook.example.org/otterbook/login
  __FORM_TOKEN=d157f83e443347c3a36efe1f (hidden readonly)
  referer=                                (hidden readonly)
  user=                                    (text)
  password=                               (password)
  <NAME>=Login                            (submit)

```

所以我们需要填充的字段分别叫做user和password。

在WWW::Mechanize中，我们可以这样来调用submit_form()方法：

```

use WWW::Mechanize;
use Readonly;

Readonly my $loginurl => 'http://otterbook.example.org/otterbook/login';
Readonly my $revurl =>
  'http://otterbook.example.org/otterbook/wiki/ReviewerLocation';
Readonly my $user    => 'username';
Readonly my $pass    => 'password';

my $mech = WWW::Mechanize->new();
$mech->get($loginurl);
$mech->submit_form(
  form_number => 2,
  fields      => { user => $user, password => $pass },
);

```

这样submit_form()就会选中正确的表单，填充需要的字段，然后提交表单（和我们点击页面上的“Login”按钮等效）。然后脚本就登录到了wiki，可以开始获取受保护页面的内容了：

```
$mech->get($revurl);
```

现在我们获取了页面的内容，然后该怎么处理它呢？

第二步：解析数据

在那个页面中的HTML表格中存放的审阅者地址信息包括City、State/etc和Country三个字段。取出其中的数据有很多方法（可以参考Kevin Hemenway和Tara Calishain合著的《Spidering Hacks》，O'Reilly出版）但我们这里会采用一个我喜欢的模块，Matt Sisk写的HTML::TableExtract。因为这个模块能大大简化从表格中获取数据的工作，另外也支持很多的高级功能。最简单的用法是告诉它需要解析的字段头列表：

```

use HTML::TableExtract;

my $te = HTML::TableExtract->new( headers => [qw(City State/etc Country)] );

```

然后我们可以把网页内容交给HTML::TableExtract模块，如同第一步提到的那样：

```
$te->parse( $mech->content() );
```

一旦完成解析，我们可以一行一行地获取数据：

```
# 如果调用 rows() 时不提供任何参数，则默认使用第一个表格。
# 因为当前页面上仅有这一张表格，所以我们这么写是没问题的。
#
# $row 是指向匿名数组的引用，其中每个元素是该行各列的值。

my @reviewlocations;
foreach my $row ( $te->rows ) {
    # trac wiki 会在 HTML 表格标签末尾添加假的换行符，所以先去掉
    chomp ( @$row );
    push @reviewlocations, $row;
}
```

第三步：对数据做地理编码并画图

为了最终完成这个有趣的项目，我们还需要完成地址的图形化，这次我们会使用另外一个类似之前介绍的HTML::GoogleMaps的模块。要知道谷歌（在基于JavaScript的动态地图之外）还提供静态地图的服务。这里我们可以使用Martin Atkins的Geo::Google::StaticMaps模块来访问这个服务。此模块的文档假定读者已经弄懂了谷歌的API文档，所以在开始使用之前请先阅读<http://code.google.com/apis/maps/documentation/staticmaps>上的资料。另外一个要使用的模块是Tatsuhiko Miyagawa的Geo::Coder::Google模块。这两个模块都需要我们提供已经进行地理编码过的数据。而之前使用的Geo::Coder::Google已经内置了地理编码的逻辑，所以省去了我们的力气。这个例子里，我们需要自己完成地理编码的过程，而它是通过下面这个例程实现的：

```
use Geo::Coder::Google;
use Geo::Google::StaticMaps;

...

sub locate {
    my $place = shift;

    # 我们可以在当前例程外部初始化该对象并将其作为参数传入例程
    my $geocoder
        = Geo::Coder::Google->new( apikey => '{your API key here}' );

    my $response;
    until ( defined $response ) {
        $response = $geocoder->geocode( location => $place );
    }

    my ( $long, $lat ) = @{ $response->{Point}{coordinates} };
```

```
    return $lat, $long;
}
```

locate()能对某个地址返回相应的经纬度。

警告：在测试这段代码的时候，我发现这个服务（或者模块）有时不能返回正确的数据，哪怕提交的查询是正确的。但如果我重复提交查询，往往能获得正确结果，哪怕是在同一个会话中。

于是我在之前的代码中做了一个比较危险的操作：如果不能获得正确的结果，就重新查询，直到返回正确坐标为止。如果你自己在处理不确定好坏的地址的时候，请不要效仿我在这里的操作。

让我们看看如何使用这个模块。第一步是构造一个需要在地图上定位的位置列表。然后只要把它们一并传递给某个方法就行了。下面的代码就是用来实现这个功能的：

```
my @markers;
# 创建一个哈希列表，每个哈希都包含着坐标点的相关信息（诸如经纬度、大小等等）
foreach my $location (@reviewlocations) {
    push @markers, {
        point => [ locate( join( ',', @$location ) ) ],
        size => 'mid' };
}

my $url = Geo::Google::StaticMaps->url(
    key      => '{your API key here}',
    size     => [ 640, 640 ],
    markers => [@markers],
);
```

调用url()会获得这样的超长URL：

```
http://maps.google.com/staticmap?format=png&
markers=42.389121,-71.097145,midred%7C34.052187,-118.243425,midred%7C39.951639,
-75.163808,midred%7C35.231402,-80.845841,midred%7C42.503450,-71.
207985,midred%7C40.567095,-105.077036,midred%7C42.375392,-71.118487,
midred%7C33.754487,-84.389663,midred%7C32.718834,-117.163841,midred%7C49.203705,
-122.914588,midred%7C50.940664,6.959911,midred%7C-33.867139,151.207114,
midred%7C37.775196,-122.419204,midred%7C37.369195,-122.036849,midred%7C42
.886875,-78.877875,midred%7C61.216583,-149.899597,midred%7C47.350102,
7.902589,midred%7C33.179521,-96.492980,midred%7C49.263588,-123.138565,
midred%7C44.250871,-79.604822,midred%7C42.125291,-71.102576,
midred%7C45.423494,-75.697933,midred%7C37.279132,-121.956295,midred%7C51
.500152,-0.126236,midred%7C32.055400,34.759500,midred%7C39.762445,
-84.205247,midred%7C50.087811,14.420460,midred%7C52.663857,-8.626773,
midred%7C43.670233,-79.386755,midred%7C42.540904,-76.658372,midred%7C32.
832207,-85.763611,midred&key=YOURKEY&size=640x640
```

而且不论你相信与否，只要你把这个URL复制到浏览器，就能看到这本书的审阅人员的住址（如图14-4所示）。

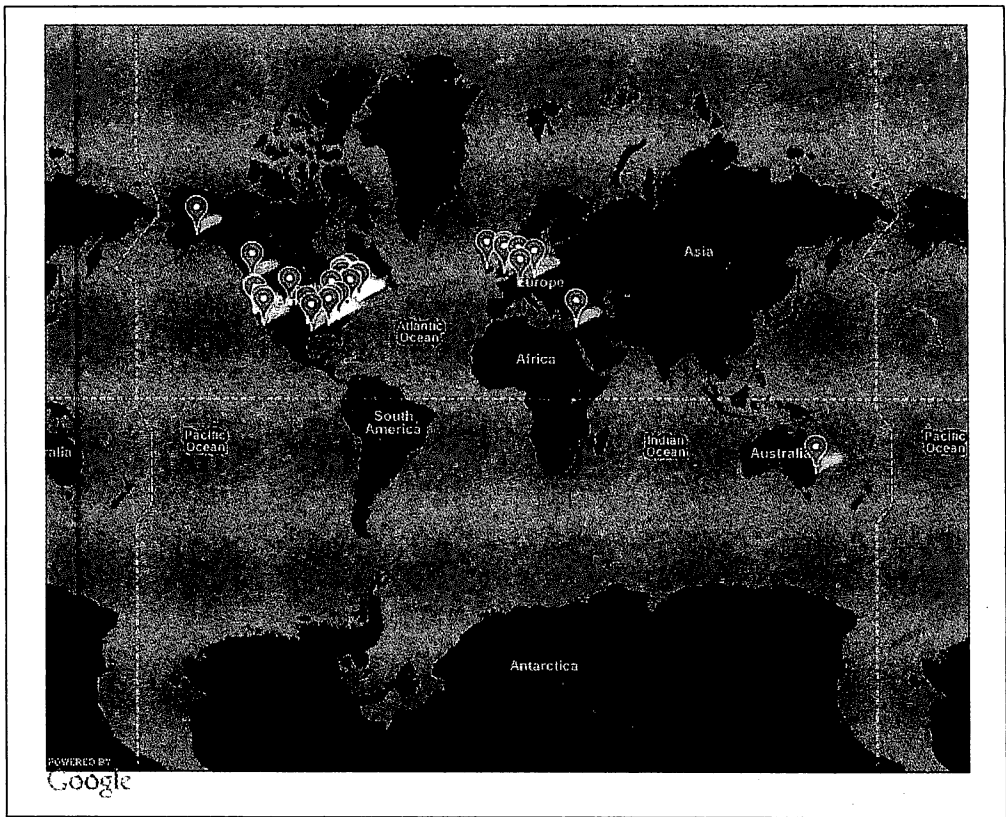


图14-4：显示所有本书审阅人员地理位置的地图

如果你希望对输出做进一步的控制（比如只希望显示北美/美国的审阅者的地址），那么可以再加上几个字段，这样可以定义地图中心和缩放级别：

```
my $url = Geo::Google::StaticMaps->url(  
    key    => '{your API key here}',  
    size   => [ 640, 640 ],  
    markers => [@markers],  
    center => [ locate('Kansas, US') ],  
    zoom   => 3,  
);
```

图14-5 显示的是居中并放大过的地图。

到此为止，我们的范例就结束了，书终于可以提交给出版社了。



图14-5：显示美国的审阅者地理位置的居中并放大的地图

总结：我们可以学到什么？

这个项目演示了：

- 如何使用WWW::Mechanize（这在当今特别需要，因为很多系统管理工具和应用都开始普及Web界面了）
- 如何使用HTML::TableExtract（用来简化Web页面的表格数据解析）
- 如何使用另外一种地理编码和地图绘制的API
- Perl社区，尤其是那些愿意帮助修改本书的人真是无处不在，而且全都充满了爱心

记住娱乐

要知道我还考虑过更多可以写在这一章的项目，因为数量太多没有办法在此介绍，但是你可以考虑自己动手去完成。包括：

- 从照片中解析EXIF元数据
- 通过家里的宽带连接来用SNMP查询连至电脑的温度传感器，并且把气温波动情况用RRDtool展现出来
- 试着读取条形码扫描器的结果
- 用CDDDB来识别CD
- 使用Perl控制家里的电器（就像MisterHouse项目那样，参考<http://misterhouse.sourceforge.net>）
- 通过电脑来产生人声（可以考虑使用Win32 SAPI），或者识别语音
- 软件包跟踪
- 并行处理
- 云计算

所有这些项目都能推动你学习到非常棒的知识 and 技能，而且可以立刻应用在自己的工作中。如果你能坚持这种对Perl的把玩以及对所有系统管理相关知识的渴望，那么相信你也从容提升各个方面的生活质量。

记住，娱乐是王道。

本章所用模块

模块名	CPAN ID	版本
Schedule::Cron::Events	PKENT	1.8
IO::File（位于IO模块内）	GBARR	1.2301
XML::Writer	JOSEPHW	0.606
File::Slurp	DROLSKY	9999.13
Time::Local	DROLSKY	1.1901
POSIX（随Perl发布）		1.15
XMLRPC::Lite（位于SOAP-Lite模块内）	MKUTTER	0.710.6
URI::Escape（位于URI模块内）	GAAS	3.29

模块名	CPAN ID	版本
XML::Simple	GRANTM	2.18
LWP::Simple (位于libwww/LWP模块内)	GAAS	5.810
HTML::GoogleMaps	NMUELLER	10
Net::DNS	OLAF	0.64
Text::CSV_XS	HMBRAND	0.58
Geo::IP	BORISZ	1.36
XML::RSS	SHLOMIF	1.42
MP3::Info	DANIEL	1.24
File::Find::Rule::MP3Info	KAKE	0.01
File::Find::Rule::Permissions	DCANTRELL	1.3
WWW::Mechanize	PETDANCE	1.52
Readonly	ROODE	1.03
HTML::TableExtract	MSISK	2.10
Geo::Google::StaticMaps	MART	0.1
Geo::Coder::Google	MIYAGAWA	0.03

本章中的资料来源

本章中的大部分资料采用自我在USENIX协会的《login》杂志 (<http://www.usenix.org/publications/login/>) 上所发表的专栏，并做了内容上的扩充。

8分钟XML教程

XML (eXtensible Markup Language, 可扩展标记语言) 的学习中最令人惊讶的地方在于它是多么容易开始。这个附录会介绍一些最常用的信息, 为你解答在学习遇到的问题。第6章的参考资料中有很多非常好的资源, 相信它们可以带你了解更多信息。

XML是一个标记语言

受益于HTML这个无处不在的老大哥, XML这一类的标记语言也逐渐被众人了解。和HTML相同, XML也是由纯文本组成的, 只是其中有一些特殊的描述性或者指令性的文本。HTML中对标记性的文本, 也称为标签 (tag), 有所限制, 然而XML却允许你自己定义标签。

于是XML提供了远超过HTML的表达能力。在第6章中能看到它的表达能力, 不过即使你从来没有接触过它, 从下面的例子中, 你也能看出它的优势:

```
<hosts>
  <machine>
    <name> quiddish </name>
    <department> Software Sorcery </department>
    <room> 314WVH </room>
    <owner> Horry Patter </owner>
    <ipaddress> 192.168.1.13 </ipaddress>
  </machine>
  <machine>
    <name> dibby </name>
    <department> Hardware Hackery </department>
    <room> 310WVH </room>
    <owner> Harminone Grenger </owner>
    <ipaddress> 192.168.1.15 </ipaddress>
  </machine>
</hosts>
```

XML有些挑剔

在喜爱XML灵活性的同时，你也要忍受它的挑剔。因为你的数据必须满足一些语法和规则，具体的约束在位于<http://www.w3.org/TR/REC-xml/>的XML规范中定义了。这里我并不打算重复官方的说明，只会建议你参考被人解读过的版本，比如Tim Bray的注解版（可以参考<http://www.xml.com>）和Robert Ducharme所著的《XML: The Annotated Specification》（Prentice Hall 出版）一书。前者在网上即可免费阅读，后者则有很多XML范例代码。

下面两个XML规则对于熟悉HTML的人来说有些特殊：

- 任何开始的东西都必须有结束。在前面的例子中，我们是用<machine>标签开始罗列机器信息的，所以也必须使用</machine>来结束它。省略结束标签对XML来说是严重的问题。
- 在 HTML 中， 这样的标签可以独立存在，但这对于XML来说是非法的。如果必须省略结尾，那么也可以这样写：

```
 </img>
```

或者：

```

```

在标签末尾额外的斜线使得 XML 解析器知道这个标签同时代表了开始和结束。一对开始和结束标签（以及其中的数据）被称为一个元素。

- 开始和结束标签必须严格匹配。大小写的差别是不被认可的，因为 XML 是区分大小写的。如果开始标签是<MaChIne>，那么结束标签就只能是</MaChIne>，而</MACHine>或者其他的变化都是不被认可的。HTML 在这方面就宽松多了。

以上就是 XML 规范中的基本规定。但是有时候你还希望能自己定义一些规则，让 XML 解释器帮你加入其他的约束（这里的“其他约束”意味着 XML 解析器可以对不合法数据“发出警告”或者“停止解析”）。用刚才的主机数据库 XML 文件作为例子，可能需要规定每个<machine>条目都必须有<name>和<ipaddress>元素。也可以规定元素的取值必须在某个范围内，比如只能是 YES 或者 NO。

这些规则的定义相比之下就显得比较复杂了，因为有好些互相补充又互相竞争的规范可供选择。

目前的 XML 规范继承了 SGML 的文档类型定义（Document Type Definition，DTD）。下面的范例 XML 文档的开头就是相关规则的定义：

```

<?xml version="1.0" encoding="UTF-8" ?>
<!DOCTYPE greeting [
  <!ELEMENT greeting (#PCDATA)>
]>
<greeting>Hello, world!</greeting>

```

范例的第一行规定了 XML 文档的版本和字符编码（Unicode）。紧接着的三行定义了文档中的数据类型。最后的一行（`<greeting>` 元素）才是实际内容的文档。

如果需要定义这个附录开头处的 XML 文件中的 `<hosts>` 相关的约束，我们可以在文件头加上这样几行：

```

<?xml version="1.0" encoding="UTF-8" ?>
<!DOCTYPE hosts [
  <!ELEMENT hosts      (machine)*>
  <!ELEMENT machine    (name,department,room,owner,ipaddress)>
  <!ELEMENT name       (#PCDATA)>
  <!ELEMENT department (#PCDATA)>
  <!ELEMENT room       (#PCDATA)>
  <!ELEMENT owner      (#PCDATA)>
  <!ELEMENT ipaddress  (#PCDATA)>
]>

```

这个规范定义了 `hosts` 元素必须包含 `machine` 元素，而 `machine` 元素必须包含 `name`、`department`、`room`、`owner` 和 `ipaddress` 元素（必须以这个顺序出现）。而且这些元素都被描述为 `#PCDATA`（请参考“剩下的话题”一节的解释）。

万维网联盟（World Wide Web Consortium, W3C）还定义了另外一个叫做 *schema* 的规范，用来实现和 DTD 类似的功能，而这个规范自己也是用 XML 代码来书写的。下面的范例 *schema* 代码使用了 XML Schema 1.0 建议语法（参考 <http://www.w3.org/XML/Schema>，1.1 版本的建议语法在本书写作时还没有正式发布）。

```

<?xml version='1.0' ?>
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema">

  <xsd:complexType name="MachineType">
    <xsd:sequence>
      <xsd:element name="name" type="xsd:string"/>
      <xsd:element name="department" type="xsd:string"/>
      <xsd:element name="room" type="xsd:string"/>
      <xsd:element name="owner" type="xsd:string"/>
      <xsd:element name="ipaddress" type="xsd:string"/>
    </xsd:sequence>
  </xsd:complexType>
  <xsd:complexType name="ListOfMachines">
    <xsd:sequence>
      <xsd:element name="machine" type="MachineType"
        minOccurs="1" maxOccurs="unbounded" />
    </xsd:sequence>
  </xsd:complexType>

```

```
<xsd:element name="hosts" type="ListOfMachines" />
</xsd:schema>
```

DTD 和 schema 都很容易变得特别复杂，所以我们在这里不做介绍。如果有兴趣，请阅读那些专门介绍 XML/SGML 的书。

两个重要的 XML 术语

如果你能弄清楚两个重要的术语，那么就可以进一步深入学习 XML。首先，XML 数据如果能符合所有的语法约束（比如标签配对等等），那么可以称为是良构（well-formed）的。一般来说良构的 XML 数据中输入错误应该比较罕见。这对于那些用来做配置文件的 XML 数据来说非常有意义，比如上一节看到的主机数据库文件就是这样。

另外，如果 XML 数据能够满足我们之前介绍的那些数据定义机制，那么它可以称为是有效的（valid）。比如，若是某个数据文件遵从 DTD 的约束，那么就是有效的 XML 数据。

有效的数据文件一定是良构的，但是这句话反过来就不一定正确。很可能遇到那些看起来赏心悦目的 XML 数据，但是没有相应的 DTD 或者 schema。这种数据往往能被正确解析，但却算不上是有效的。

剩下的话题

下面的三个术语也常常在与 XML 相关的文章中出现，对初学者来说往往是障碍：

属性

这指的是在起始标签中的描述信息。拿前面的例子来说，`` 元素中的属性就是 `src="picture.jpg"`。在 XML 领域，关于何时应该使用元素，何时应该使用属性常常有争论。这方面最好的建议可以参考 <http://www.oasis-open.org/cover/elementsAndAttrs.html>。

CDATA

CDATA（Character Data，字符数据）这个术语在两个环境中会用到。大多数场合这个术语的意思是指那些没有标记（比如标签）的数据。第二个意思是指 CDATA 段落。CDATA 段落使得 XML 解析器在这里关闭它的标记识别功能。这种段落的数据往往比较奇怪，可以参考下面的例子：

```
<![CDATA[<greeting>Hello,world!</greeting>]]>
```

在这里，`<greeting></greeting>` 标签被当作纯文本来处理，也就不必解析了。

PCDATA

之前说的 Tim Bray 的 XML 规范注解里面给它下了这样的定义：

PCDATA 字面上的意思是“Parsed Character Data”。它也是从 SGML 中继承来的概念，在这里“Parsed”意味着 XML 解析器会对段落中的 < 和 & 标记进行处理。

你可以把它理解成是 CDATA 的数据再加上少数几个标记。大多数的 XML 数据可以归到这个类别中。

最后还有两个小技巧是有经验的 XML 用户牢记于心的，希望新手知道之后可以少走冤枉路：

- 有些特殊的字符是不能在 XML 数据中直接使用的，必须使用 entity（实体）的形式来间接表达，这和 HTML 一样。特殊字符包括：<、>、&、'（单引号）以及 "（双引号）。它们的表达方法和 HTML 一样：<、>、&、' 和 "。很多新手常常忘记这一点，导致数据不能被正确解析。
- 如果你需要在文档中使用非 UTF-8 数据，那么请记住声明编码。这在 XML 中的声明方法是：

```
<?xml version="1.0" encoding="iso-8859-1" ?>
```

常常遇到的问题是省略了这个声明，或者错误地把文档声明为 UTF-8 编码（而其实数据属于其他字符集）。

XML 的学习会有些难度，但这个教程应该已经能带你入门了。掌握了基础知识之后，你可以进一步学习更加复杂的规范，比如 XSLT（用来把 XML 翻译成其他格式，比如 HTML）、XPath（一种用来筛选 XML 文档内容的规范）以及 SOAP/XML-RPC（用来与远程服务进行 XML 信息交互）。

更多参考资料

请参考第 6 章结尾部分的内容，获取更多关于 XML 主题的参考资料。

10分钟XPath教程

在我们正式介绍 XPath 之前，先要澄清三个问题。

首先，为了能读懂这个附录，你需要起码懂些基本的 XML 知识。如果还没有这个基础，请参考附录 A。

其次，XPath 是一门独立的语言。要知道 XPath 1.0 规范说明书有密密麻麻的 34 页，而 XPath 2.0 的则有 118 页厚。这个附录并不打算替 XPath 进行任何功能、表达性或者复杂度方面的辩解，尤其是 2.0 版的。这里只是打算介绍那些对 Perl 程序员来说最有用的 XPath 基础知识。

最后，这个附录主要针对 XPath 1.0，因为目前为止还没有哪个 Perl 模块能支持 XPath 2.0。

在说了这么多之后，现在让我们开始讨论“什么是 XPath”这样的话题，尤其是你可能更加感兴趣的“为何我要学习它”。XPath 是一个 W3C 的规范，用来定义如何从 XML 文档中取出部分有用的数据。如果你曾经写过从 XML 文档中解析部分数据的代码，那么 XPath 能让你做得更加轻松自如。这是一个非常简洁的语言，不过功能却很强，也没有陷入追逐功能的误区。只要你能用 XPath 描述需要的数据（这并不太难），XPath 解析器就能帮你取出这些数据，或者最起码帮你的程序定位到 XML 文档的正确位置。通常这些只需要一行 Perl 代码就可以做到。

XPath 的基础概念

在正确使用 XPath 之前有些基础概念先要弄清楚，我们就从最简单的开始介绍。

基本定位路径

为了理解 XPath，你首先必须搞清楚 XML 文档应该都可以被解析成树形结构。文档的

元素（还包括其他一些东西，这里先不介绍）就是这棵树的叶子。为了帮助你理解这点，我们可以先把第6章里的样本 XML 文档拿来参考。为方便起见下面就列出它的内容：

```
<?xml version="1.0" encoding="UTF-8"?>

<network>
  <description name="Boston">
    This is the configuration of our network in the Boston office.
  </description>
  <host name="agatha" type="server" os="linux">
    <interface name="eth0" type="Ethernet">
      <arec>agatha.example.edu</arec>
      <cname>mail.example.edu</cname>
      <addr>192.168.0.4</addr>
    </interface>
    <service>SMTP</service>
    <service>POP3</service>
    <service>IMAP4</service>
  </host>
  <host name="gil" type="server" os="linux">
    <interface name="eth0" type="Ethernet">
      <arec>gil.example.edu</arec>
      <cname>www.example.edu</cname>
      <addr>192.168.0.5</addr>
    </interface>
    <service>HTTP</service>
    <service>HTTPS</service>
  </host>
  <host name="baron" type="server" os="linux">
    <interface name="eth0" type="Ethernet">
      <arec>baron.example.edu</arec>
      <cname>dns.example.edu</cname>
      <cname>ntp.example.edu</cname>
      <cname>ldap.example.edu</cname>
      <addr>192.168.0.6</addr>
    </interface>
    <service>DNS</service>
    <service>NTP</service>
    <service>LDAP</service>
    <service>LDAPS</service>
  </host>
  <host name="mr-tock" type="server" os="openbsd">
    <interface name="fxp0" type="Ethernet">
      <arec>mr-tock.example.edu</arec>
      <cname>fw.example.edu</cname>
      <addr>192.168.0.1</addr>
    </interface>
    <service>firewall</service>
  </host>
  <host name="krosp" type="client" os="osx">
    <interface name="en0" type="Ethernet">
      <arec>krosp.example.edu</arec>
      <addr>192.168.0.100</addr>
```

```

    </interface>
    <interface name="en1" type="AirPort">
        <arec>krosp.wireless.example.edu</arec>
        <addr>192.168.100.100</addr>
    </interface>
</host>
<host name="zeetha" type="client" os="osx">
    <interface name="en0" type="Ethernet">
        <arec>zeetha.example.edu</arec>
        <addr>192.168.0.101</addr>
    </interface>
    <interface name="en1" type="AirPort">
        <arec>zeetha.wireless.example.edu</arec>
        <addr>192.168.100.101</addr>
    </interface>
</host>
</network>

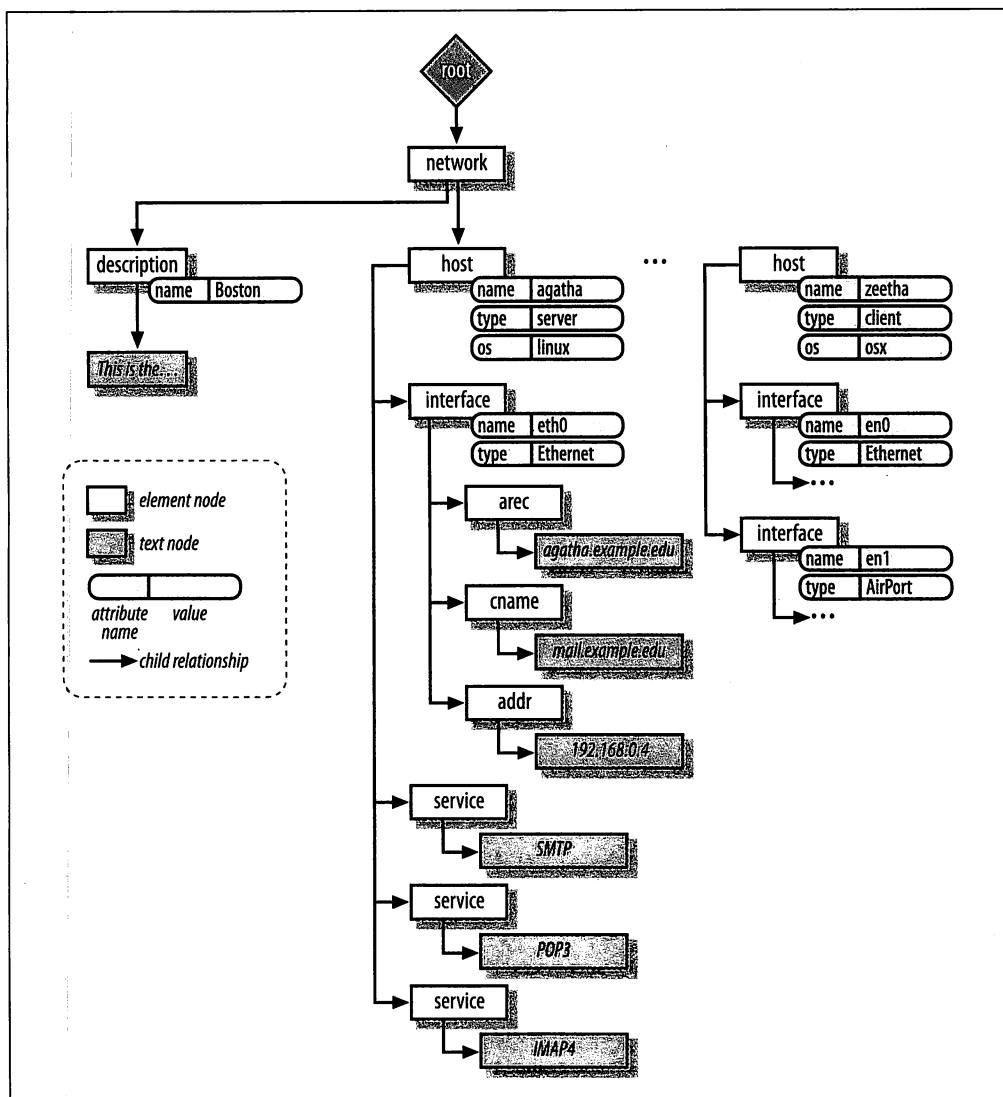
```

如果你把它解析成一棵树的话，那么看起来应该会如图B-1所示。

这棵树的根对应于文档的根元素（<network></network>），文档中的其他元素都挂在它的下面。每个元素节点下面还可以挂属性节点（可选）以及子文本节点（如果有的话）。比如对于 <element attrib="value">something</element> 这样的 XML 元素，XPath 解析器应该会生成一个 <element></element> 节点，再加上附属的 attrib 属性节点以及一个 something 文本节点。这个树状的模型对于下面的介绍非常重要，所以请务必多花些时间把图B-1牢记于心。

如果这幅图现在对你来说有点类似第2章中的树形图的话，那就行了，这正是设计的初衷。XPath 使用了定位路径（location path）的概念来描述通往文档中某个节点（或某些节点）的寻址方式。定位路径可以从树的顶端开始，称为绝对路径，也可以从树中的某个位置开始，称为相对路径。这里借鉴了文件系统的一些惯例，把开头的“/”定义为树的根节点，把“.”定义为当前路径（也可以称为上下文节点），而“..”则意味着上下文节点的父节点。

如果你愿意的话，可以把定位路径想象成通往指定节点（或一组节点）的一条路线。比如说，如果我们想去的是 <description>?</description> 节点，那么可以走 /network/description 路线。如果我们使用 /network/host 这个定位路径，那么就能到达那一层的所有 <host></host> 节点。如果想要进一步深入树的下一层，那么就需要一些机制来区分各个 <host></host> 节点。具体怎么区分就是更加深入的 XPath 专题了，这里我们先不介绍，而是专注于 XML 树的主要定位机制。



图B-1: XML 文档节点树

我们的样本文件中除了标签之外还有真正的数据。元素本身往往带有属性，比如`<interface name="en1" type="AirPort">`；或者作为数据的标签，比如`<addr>192.168.0.4</addr>`。那么如何访问这些位置的数据呢？我们可以在属性名之前加上一个@符号。比如`/network/description/@name`会得到`name="Boston"`这个数据。要获得元素的文本节点的内容，可以在定位路径的末尾加上`text()`，比如`/network/description/text()`，这会得到`This is the configuration...`这段数据。

XPath中通配符的工作原理非常类似于文件系统。`/network/host/*/arec/text()` 能用

来定位所有在 `<host></host>` 节点之下（带有 `<arec></arec>` 子节点）的元素节点^[注1]，并且返回它们的`<arec></arec>`元素的内容。在这里返回的其实是所有接口的 DNS A 资源记录。

```
agatha.example.edu
gil.example.edu
baron.example.edu
mr-tock.example.edu
krosp.example.edu
krosp.wireless.example.edu
zeetha.example.edu
zeetha.wireless.example.edu
```

属性也可以像元素那样使用`@*`这样的通配符来匹配。`/network/host/@*`能返回所有`<host></host>`元素的属性。

在进入下一节之前还有最后一个值得介绍的语法。XPath 有一个我觉得可以称为“魔力操作符”的定位路径符号。如果你在定位路径的任何位置使用双斜线（//），那么这就意味着从树的那个位置开始一直往下找，并返回所有剩余元素匹配的结果。比如，如果我们书写 `//arec/text()`，就能返回和前面的例子相同的 DNS A 资源记录结果，因为这个双斜线操作符能够从树的根节点开始搜索，并试图发现所有带有文本节点的`<arec></arec>`元素。另外，也可以在定位路径的中间使用双斜线，比如`/network//service/text()`。我们的范例文件树并不深，但是你可以想象这样的功能对于那些复杂的树来说应该是非常有用的。

谓词

在上一节中，我们很谨慎地跳过了如何识别树中同名分支的问题。在我们的范例文件中，我们在树的第三层有五个同名的 `<host></host>` 元素。虽然它们有不同的属性和数据，但这对只有元素名的定位路径并无帮助。如果我们书写 `/network/host`，这里的 `host` 就是所谓的“节点测试”。它用来选择树中的某个定位路径中的分支。但是这个例子中的所有分支都是一样的，所以无法选择分支。

这正是我们介绍 XPath 谓词(predicate)的时候。谓词让我们能够从所有可选的节点中挑出真正需要的。`/network/host` 会返回所有待选的主机节点，我们需要用某种方法对它们进行筛选。谓词是使用方括号（`[]`）来表示的。在定位路径中任何需要筛选分支的位置都可以使用谓词。

注1： 在这一章的开头，我提到过XPath能把文档解析成一组节点，其中包含元素和其他对象。这里我们可以用通配符 `*` 来匹配元素节点，而`node()`可以用来匹配所有节点（元素或者其他对象）。

最简单的谓词类似于数组下标，比如`/network/host[2]/interface/arec/text()`。这个定位路径能返回第二台主机的接口名，这里的“第二”是指文档顺序的第二个。如果你站在主机节点的位置不知道要往哪里走的时候，谓词就会告诉你：“请选第二个分支。”

警告： Perl 程序员应该对这个下标语法不陌生，不过可能还是有些不习惯。因为这里的数组下标是从 1 开始的，而不是 0。

如果在谓词中只能使用下标值的话，那就显得太无聊了。谓词这个概念的引入是为了带来更加丰富的语法，XPath 正是因为有了它才变得非常有用。典型的复杂谓词看起来像这样：`/network/host[@name="agatha"]`。这里我们通过一个特殊的属性值来完成对`<host></host>`的筛选。^[注2]

谓词也不必总是用在定位路径的末尾，你还可以在定位路径的中间使用它。如果我们搜索所有 Linux 服务器的名字，可以使用`/network/host[@os="linux"]/service/../@name`这样的定位路径。这里的`os`属性被特别用来作为谓词，筛选所有的`<host></host>`元素，看看它们哪些拥有 `linux` 这个值。然后定位路径继续查找那些分支的`<service></service>`节点（也就是那些作为服务器的主机）。在这里我们已经到达了`<service></service>`节点，所以可以使用`../@name`来获取父节点的`name`属性（也就是那些有`<service></service>`子节点的`<host></host>`节点的名字）。

我们可以这样来匹配某个节点的内容：`//host/service[text()='DNS']`。这个定位路径会从树的根节点开始寻找嵌入在`<host></host>`节点中的`<service></service>`节点。对于所有符合以上条件的分支，再查看其中是否有“DNS”这样的内容。

这里的定位路径还能进一步简化。你只要使用一个“.”（点号），就可以省略`text()`调用。这个点号能指示解析器去比较当前节点的内容。

等值比较只是所有比较操作的一种。我们的范例数据无法演示其他的比较，不过确实可以使用`[price > 31337]`这样的谓词来筛选节点。

现在这个定位路径看起来越来越像真正的计算机语言了，不是吗？随着函数的引入，这个概念会越来越清晰。XPath 定义了众多的函数用来操作节点集合、字符串、布尔值以及数值。实际上我们已经看到了不少函数，因为`/network/host[2]/interface/arec/text()`的意思其实是`/network/host[position()=2]/interface/arec/text()`。

注2： 在我们进一步介绍谓词之前，有必要介绍一个不太明显的问题。如果定位路径有问题的话（比如我们要找的是 `/network/admin/homephonenumber`），程序并不会停下来报告错误，只是不返回任何值。

下面的例子应该能给你留下更深的印象，这是一个用来筛选 HTTP 和 HTTPS 服务节点的定位路径（也允许服务名周围包含空白）：`//host/service[starts-with(normalize-space(.),'HTTP')]`。这个`starts-with()`函数顾名思义能对当前节点的内容进行比较，如果它是以第二个参数提供的字符串开始的，则返回真。XPath 规范定义了很多的函数，只是对初学者来说比较难用。在网上搜索“XPath 谓词”应该能发现很多对于规范的进一步解释。

简写和坐标轴

这个附录是从最简单的 XPath 核心概念开始的，每一小节都逐渐加入了一些更复杂的概念。这里介绍的是关于定位路径的最后一一些细节。我们要提醒的是之前看过的所有定位路径其实都是规范中所谓的“简写语法”。非简写的语法在大多数时候是用不到的，但是在某种特殊情况下就变得特别需要。这里我们会快速介绍它们，这样在需要的时候你就知道确实有这样的语法可用。

那么我们已经介绍过的定位路径中都有哪些简写呢？比如`/network/host[2]/service[1]/text()`，它完整的意思是：

1. 从树的根节点开始。
2. 往树的子节点方向（也就是下方）走，查找名为 `network` 的元素。
3. 到达 `<network></network>` 节点。这里成为上下文节点。
4. 再往树的子节点方向走，查找名为 `host` 的元素。
5. 到达树的多个 `<host></host>` 节点那一层。筛选第二个位置的节点，使这里成为上下文节点。
6. 再往树的子节点方向走，查找名为 `service` 的元素。
7. 到达树的多个 `<service></service>` 节点那一层。筛选第一个位置的节点，使这里成为上下文节点。
8. 再往上下文节点相关的文本节点走，到此为止。

如果我们把以上的定位路径用完整的语法来书写，看上去应该像这样（字符太多，被分为两行）：

```
/child::network/child::host[position()=2]/child::service[position()=1]/
child::text()
```

我们加入的主要是所谓的“坐标轴”（axis的复数形式，这里的 `axes` 并非斧子的意思）。这个定位路径的每一步都加入了坐标轴，用来指明解析器要往上下文节点的哪个

方向走。我们总是告诉它跟随 `child::` 方向，也就是说往上下文节点的子树方向走。在面对文件系统的时候，我们非常习惯于自上而下的语法，所以在面对 `/dir/sub-dir/file` 这样的语法的时候并不会困惑。这也就是为什么 XPath 的简写语法看上去那么自然。但是 XPath 并没有约束我们只能往树的子节点方向移动。之前的例子中，我们看到了 `//` 语法。在书写 `/network//cname` 的时候，我们的意思其实是 `/child::network/descendant-or-self::cname`。也就是：

1. 从树的根节点开始。
2. 往子节点的方向找 `<network></network>` 节点（也可能是许多节点）。在找到之后，使它成为上下文节点。
3. 在上下文节点中查找，或者再不断地深入子树，直到我们找到 `<cname> </cname>` 节点（也可能是许多节点）。

另外三个我们已经知道如何简写的坐标轴是：`self::(.)`、`parent::(..)` 以及 `attribute::(@)`。另外还有八个坐标轴没有简写：`ancestor::`、`following-sibling::`、`preceding-sibling::`、`following::`、`preceding::`、`namespace::`、`descendant::` 以及 `ancestor-or-self::`。

其中，`following-sibling::` 可能是最有用的坐标轴了，我们可以看一个相关的例子（这个附录的参考资料中列出了其他坐标轴的描述信息）。`following-sibling::` 坐标轴能让解析器分析同一级别的其他元素，也就是上下文节点的兄弟元素。如果我们需要定位那些带有多个接口的主机，可以这样来写（还是长长的单行代码）：

```
/child::network/child::host/child::interface/following-sibling::interface/  
parent::host/attribute::name
```

这段代码的大意是：“从 `network` 节点往下走，直到发现带有 `interface` 节点的 `host` 节点，然后检查它是否还有同级的兄弟节点。如果确实有，那么返回到 `host` 节点，并同时返回它的 `name` 属性。”

进一步探索

如果你发现 XPath 真的很有趣，那么可以进一步深入学习它，在这一章以外你肯定能找到更多的参考资料。你可以阅读规范说明书，也可以参考下一节列出的资料，自学其他的谓词和坐标轴。另外还可以提前学习 XPath 2.0，这样一旦有 Perl 模块的支持，你就可以充分使用它了。不论如何，请常常用这个语言做些练习，这样你手上就有了另一个好用的工具。

更多参考资料

<http://www.w3.org/TR/xpath> 和 <http://www.w3.org/TR/xpath20> 是 XPath 1.0 和 2.0 规范的官方网站地址。我建议你阅读过一两本好的教程（如下面所列的教程）之后再去看这些规范。

《XML in a Nutshell》（Third Edition），由 Elliotte Rusty Harold 和 W. Scott Means 所著（O'Reilly 出版），以及《Learning XML》（Second Edition），由 Erik T. Ray 所著（O'Reilly 出版），这两本书中都有关于 XPath 的极好的内容。在目前我所阅读过的相关教程中，这两本是最好的。

<http://www.zvon.org/xxl/XPathTutorial/General/examples.html> 是一部大部分由定位路径范例及这些定位路径如何映射到范例文档所组成的教程。如果你习惯于照着例子依样画葫芦的方式学习，这个资源对你能有所帮助。

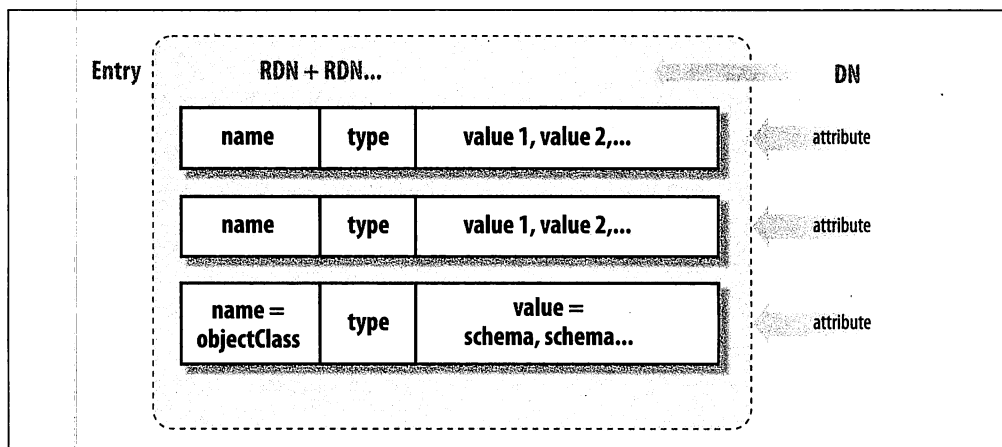
有很多工具可以让你输入一个 XPath 表达式并在样本文档中显示返回内容。有些解析器（如 *libxml2* 解析器）甚至自带了提供该功能的工具。你应该有这样一个工具，因为在创建和调试定位路径的时候它们是非常有用的。我大部分时候都用 Oxygen XML 编辑器内置的工具(<http://www.oxygenxml.com>)。

另外一个用 XPath 处理 XML 文档的很酷的工具是 XSH2 (<http://xsh.sf.net>)，由 Petr Pajas 所开发，这位作者也是 XML:LibXML 模块当前的维护者。通过它我们用 XPath 1.0 对 XML 文档进行操作就像操作文件系统路径一样简单。

10分钟LDAP教程

轻量目录访问协议（Lightweight Directory Access Protocol，LDAP），是用来访问日渐普及的目录服务的协议^[注1]。随着时间的推移，系统管理员正逐渐且越来越多地与 LDAP 服务器及客户机打交道。例如活动目录（Active Directory）和 Mac OS X 的开放目录都是基于 LDAP 的实现。这个教程会介绍 LDAP 的术语和概念，以便你理解第 9 章的内容。

对 LDAP 的操作主要发生在一种叫做条目（entry）的数据结构上。图C-1是我们理解条目时可以参考的概念图。



图C-1：LDAP 条目数据结构

条目带有一系列称为属性的成员，用来存储条目的具体数据。从数据库的角度来说，这有些类似于记录中包含的多个字段。在第 9 章中，我们使用 Perl 来存取 LDAP 目录中的主机列表。主机条目主要的属性有：name、model、location、owner 等。

注1： 特别强调一下：LDAP是一个协议，而不是关系数据库。这个协议是用来和有些类似数据库的目录服务通信。数据库和目录服务之间的差异可以参考第9章。

除了名字以外，属性还有自己的类型和值。值必须符合类型的定义。比如对于员工信息来说，条目的 `phone` 属性的类型应该是 `telephoneNumber`，而它的值应该就是员工的电话号码。类型也决定了值的存储内容（字符串、数字等）、排序机制、搜索方法（是否区分大小写）等等。为了容纳多个值，你可以把它们都存放在一个条目中。也就是说在某个条目中设置多个成员属性，每个属性中存放一个成员值。

条目的内容和结构是由它的对象类定义的。对象类（连同服务器和用户设定）指定了条目中哪些属性是必需的，哪些是可选的。每个条目都可以属于多个对象类，这时候对象类的声明会合并起来。与条目相关的对象类是使用名为 `objectClass` 的特殊属性来记录的。

让我们进一步查看 `objectClass` 属性，因为它打开了通往其他 LDAP 术语的道路。如果我们仔细考虑 `objectClass` 属性，就会发现下面的事实：

LDAP 是面向对象的。

在 `objectClass` 属性中的每个值都是一个对象类的名字。前面说到，这些类要么定义了条目中必需的属性和可选属性，要么在此基础上扩展其他类的定义。

我们可以看看下面的例子。假如某个条目的 `objectClass` 包含了 `residentialPerson` 字符串，而这个对象类在 RFC 2256（全名是 A Summary of the X.500(96) User Schema for Use with LDAPv3）中的定义如下：

```
residentialPerson
( 2.5.6.10 NAME 'residentialPerson' SUP person STRUCTURAL MUST 1
MAY ( businessCategory $ x121Address $ registeredAddress $
destinationIndicator $ preferredDeliveryMethod $ telexNumber $
teletexTerminalIdentifier $ telephoneNumber $
internationalISDNNumber $
facsimileTelephoneNumber $ preferredDeliveryMethod $ street $
postOfficeBox $ postalCode $ postalAddress $
physicalDeliveryOfficeName $ st $ 1 ) )
```

这里的定义声明了 `residentialPerson` 对象类的条目都必须带有 `l`（“locality”的简写）属性，以及很多的可选属性（`registeredAddress`、`postOfficeBox` 等等）。这个声明中最重要的部分就是 `SUP person` 字符串，它的意思是 `residentialPerson` 会从 `person` 对象类中继承属性。`person` 类的定义如下：

```
person
( 2.5.6.6 NAME 'person' SUP top STRUCTURAL MUST ( sn $ cn )
MAY ( userPassword $ telephoneNumber $ seeAlso $ description ) )
```

所以，`residentialPerson` 对象类的条目必须带有 `sn`（surname）、`cn`（common name）和 `l`（locality）属性，而可选的属性也是这两个 RFC 规范的 `MAY` 区段的合

并。我们也了解到`person`类的父类是`top`，所以它的子类`residentialPerson`是最接近顶层的类。

大多数情况下，我们都会使用预定义的标准对象类。如果确实需要构造自己的类，可以从已经定义的类中找到最接近的对象类，以此为基础进一步构造，类似于`residentialPerson`在`person`基础上扩展。

LDAP来自于数据库。

从`objectClass`中还能看到LDAP的数据库本质。某个条目对应的所有对象类被称为`schema`。之前引用的RFC就是LDAP schema规范的例子。我们并不会进一步介绍和`schema`有关的内容，因为`schema`设计（类似于数据库的方案设计）是一个大的专题，值得写一整本书。但是起码你会对“`schema`”这个名词越来越熟悉，因为我们会常常提到它。

LDAP并不局限于存储树状数据。

最后一点要提及的会有助于我们看到LDAP的整个画卷。要注意的是，虽然在前面的例子中我们看到了`top`对象父类的声明，但是这并不意味着简单的自顶而下的结构。因为我们还可以设置对象的别名（`alias`）。如果为条目设置了`alias`，那么这个条目就是另一个条目的别名（通过`aliasedObjectName`属性来声明）。LDAP的定义虽然强烈建议树形的继承结构，但是并非必须。在实际应用的时候要特别注意这一点，免得落入继承关系的陷阱。

LDAP 数据组织

到目前为止我们一直在讨论单个条目，但是目录中不可能只有一个条目。如果我们开始考虑一个充满条目的目录时，很快会面对一个问题，如何找到某条数据呢？

我们之前讨论的所有概念在LDAP规范中称为“信息模型”，这个模型关注的是如何表达信息。而刚才提出的问题则在LDAP规范的“命名模型”中解答，这个模型关注的是如何组织信息。

如果你回头看看图C-1，你就会发现我们讨论了条目的各个方面，除了它的名字。其实每个条目都有一个名字，那就是它的标识名（Distinguished Name, DN），DN是由一系列的相对标识名（Relative Distinguished Name, RDN）组成的。我们稍后会介绍DN，先仔细看看RDN这个组成部分。

RDN是一个或多个属性的名-值对。比如`cn=Jay Sekora`（其中的`cn`代表“common name”）就可以算是一个RDN，属性名是`cn`，而值是`Jay Sekora`。

无论是LDAP还是X.500都没有规定在RDN中必须使用哪些属性，但是它们都要求

RDN 在目录的每一层内必须是唯一的。这个限制的存在是因为 LDAP 无法用“目录树的第四个分支的第三个条目”这样的寻址方法来访问，而必须用唯一的名字来区分条目。让我们看看这个约束对我们有什么影响。

看看我们曾经使用的 RDN: `cn=Robert Smith`，这应该不算是一个好的 DSN，因为哪怕是很小的机构中也可能会有不止一个 Robert Smith。如果你的机构人数众多，而目录的层次又不深，那么像这样的名字冲突会更常见。有个相对好些的选择是使用两个属性组合作为 RDN，比如 `cn=Robert Smith + l=Boston`（RDN 的属性可以通过加号组合）。

我们的组合 RDN 加入了位置属性，不过仍然可能遇到问题。我们做的只是推迟了名字冲突的发生，并没有完全避免。另外，如果 Smith 调去了另一个分支机构，那么我们必须同时修改他的位置属性和 RDN。可能最好的 RDN 应该是此人唯一并保持不变的用户 ID，比如邮件地址的用户名部分（那么 RDN 就应该是 `uid=rsmith`）。这个例子让我们了解在设计 schema 的时候是如何挑选属性的。

敏锐的读者可能注意到我们还没有真的扩展到整个画卷，仍然在讨论单个条目的话题。但是 RDN 其实是一个跳板，从这里我们可以进入类似树结构的世界^[注2]。这个结构往往称为目录信息树（Directory Information Tree, DIT），或者干脆称为目录树。而目录树的称呼应该更加稳妥，因为在 X.500 的名词中 DIT 指的是全球信息树，类似于全球的 DNS 目录或者全球管理信息库（MIB，我们在附录G中讨论 SNMP 时介绍）。

让我们把 DN 放到画卷中。目录树中的每个条目都可以使用唯一的名字来访问。DN 是由 RDN 的序列组成的（中间以逗号或者分号连接），这个序列能够指明去往根条目的道路。如果我们跟随图C-2中箭头的指引并积累 RDN，就能构造出特定条目的 DN。

在第一个图片中，我们的 DN 应该是：

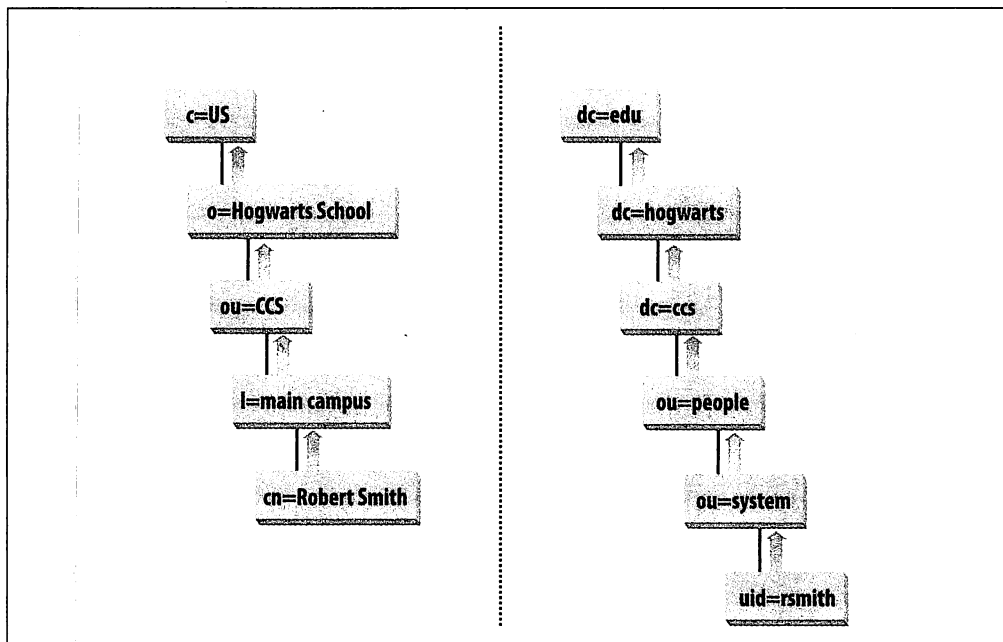
```
cn=Robert Smith, l=main campus, ou=CCS, o=Hogwarts School, c=US
```

在第二个图片中，它是：

```
uid=rsmith, ou=system, ou=people, dc=ccs, dc=hogwarts, dc=edu
```

`ou`是organizational unit的简称，而`o`则是organization的简称，`dc`代表“domain component（域组件）”，也就是 DNS，而`c`则是 country的简称。

注2：我这里之所以讲“类似树”而不是直接称之为“树”，是因为之前提到的 `alias` 对象类可以构造一个并不严格的树形数据结构（至少从计算机科学的角度来看，这属于有向非循环图（directed-acyclic-graph）的概念范畴）。



图C-2：回归至树根来产生 DN

一个关于 DN 的比喻是“文件系统的绝对路径名”，不过 DN 更加类似邮政地址，因为它的顺序是最细节的条目最先。邮政地址类似下面这样：

Doreen Hodgkins
288 St. Bucky Avenue
Anywhere, MA 02104
USA

你是从最细节的对象（人）开始，然后是更加宽泛的对象，最后达到最宽泛的对象（国家），DN 也是如此。

你可以在 DN 里面看到同样的顺序。目录树的顶端称为目录的后缀，因为它是目录树中所有 DN 的终点。后缀在使用多服务器代理来构造多层架构的时候很重要。使用 LDAPv3 的 *referral*（推介）的时候，可以在目录树的某个条目上声明“对于这个后缀的所有条目，请参考那个服务器”。推介是使用 LDAP URL 来指定的，看上去类似于网站的 URL，区别是它是指向某个 DN 或者其他 LDAP 相关的信息。下面的例子就是 RFC 2255 中的 LDAP URL 格式：

```
ldap://ldap.itd.umich.edu/o=University%20of%20Michigan,c=US?postalAddress
```

另外还可以在客户机/服务器验证的过程中使用目录后缀，因为客户机通常是连接到服务器的某个目录树，所以它就通过后缀“绑定”到这个服务器。我们会在第 9 章中看到这个过程的具体步骤。

至此你已经了解 LDAP 的数据组织方式以及相关的术语。有了这些基础知识，在第 9 章中的那些介绍应该会更加清晰了。

15分钟SQL指南

关系数据库可以成为系统管理的绝佳工具，而结构化查询语言（SQL）是访问和管理关系数据库的语言。所以对系统管理员来说，学习 SQL 也是非常必要的。这个附录就是为了让你具备数据库开发的能力，甚至可以让你具有一些管理能力，能够使你免于多年的摸索。当然你不必真的完全了解所有的 SQL，但是一旦有人给你一段代码的时候，你应该能读懂它。而且你有必要知道如何进一步深入学习。这对我们在第7章介绍的集成 SQL 和 Perl 的主题来说，是必要的基本功。

SQL 是一个用来处理关系数据库（及其组件）的通用语言，而表就是最常见的一种组件。表的行列结构使它看上去非常类似于电子表格，但是这个相似只是表面的。表中存储的元素并没有任何依赖关系，也就是说表中不能存储某个计算公式，只能存放原始数据。大多数 SQL 语句都是用来处理表中的行数据和列数据的，通过它们用户能添加、删除、修改数据，也可以进行查询、排序或者和其他表的数据进行关联。

让我们看看 SQL 中提供的操作符。如果你需要测试这些操作符，就需要登录某种 SQL 数据库。如果你能访问某种商业数据库，比如 Oracle、Sybase、IBM、微软或者其他供应商的产品当然不错。如果没有这些环境，也可以下载卓越的开源数据库 MySQL（<http://www.mysql.org>），或者考虑使用更加简单的无服务器开源数据库（<http://www.sqlite.org>）。

在这个附录中，我们会尽量使用通用 SQL，不过每个数据库服务器都有自己的 SQL 方言。特定数据库实现的 SQL 语句会特别注明。

这里的 SQL 代码会遵从大多数 SQL 书籍中遵循的大写格式。也就是说语句中的那些保留字都会使用大写。

附录里大多数的 SQL 代码都会使用第 5 章中的平面文件主机数据库表。请参考表D-1来了解数据的格式。

表D-1：我们的主机表

名字	IP地址	别名	所有者	部门	大厦	房号	制造商	机型
shimmer	192.168.1.11	shim shimmy shimmydoodles	David Davis	Soft- ware	Main	909	Sun	M4000
bendir	192.168.1.3	ben bendoodles	Cindy Coltrane	IT	West	143	Apple	Mac Pro
sander	192.168.1.55	sandy micky mickydoo	Alex Rollins	IT	Main	1101	Dell	Optiplex 740
sulawesi	192.168.1.12	sula su-lee	Ellen Monk	Design	Main	1116	Apple	Mac Pro

创建/删除数据库和表

起初，数据库服务器中并没有任何可用的数据对象。所以我们需要创建自己的数据库：

```
CREATE DATABASE sysadm ON userdev=10 LOG ON userlog=5
GO
```

这个 SQL 语句能在 userdev 设备上创建一个 10 MB 大小的数据库，以及 userlog 设备上的 5 MB 大小的日志文件。这个语句是针对 Sybase/Microsoft SQL Server 的，其他的数据库服务器有不同的数据库创建语法。^[注1]这里我们把数据库创建在一个预定义的存储设备（为数据库服务器预先分配的存储空间）上，并且把日志信息（有关数据操作的信息以及其他维护信息）存储在另一个专用的设备上。

注意： 这里的 GO 命令用来告诉某些交互式数据库客户端何时可以执行之前输入的命令。（这些客户端往往提供一些 SQL 以外的管理命令，用来简化和数据库的交互，比如 MySQL 的 DESCRIBE 命令能用来显示表结构）。这个命令并非 SQL 语句的一部分。其他的数据库客户端可能要求你在每条语句的最后输入一个分号。在以后的范例中，我们假定你总是记得在每条语句的后面输入 GO 或者分号来满足客户端的需要。另外，我们也会使用 -- 来作为嵌入在 SQL 代码中的注释。

要删除这个数据库，我们可以使用 DROP 命令：

```
DROP DATABASE sysadm
```

现在让我们来创建一个空表，用于存储表D-1中的数据：

```
USE sysadm
-- 最后一次提醒：如果你在使用交互式数据库客户端，此处每条语句你需要自行输入 GO 或者 ；
```

注1： 实际上每种数据库都有不同的“数据库”定义。这个术语对 Oracle DBA 来说意义更加丰富，而对 MySQL DBA 则相对简单。

```
CREATE TABLE hosts (
  name      varchar(30)    NOT NULL,
  ipaddr    varchar(15)    NOT NULL,
  aliases   varchar(50)    NULL,
  owner     varchar(40)    NULL,
  dept      varchar(15)    NULL,
  bldg      varchar(10)    NULL,
  room      varchar(4)     NULL,
  manuf     varchar(10)    NULL,
  model     varchar(10)    NULL
)
```

首先，我们指定需要使用的数据库（sysadm）。USE 语句必须在其他所有语句之前运行，它也是一条独立的语句。

然后，我们指定需要创建的表名，以及每个字段的名称、数据类型/长度、NULL/NOT NULL 设定。这里让我们先介绍一些数据类型。

数据库中 can 存放几种类型的数据，比如数字、日期、文本，甚至还有图像和其他二进制类型。在创建字段的时候，需要指定数据的类型。我们的例子比较简单，所以表中主要是简单的varchar字段（也就是不需要用空格来补齐的字符串）。有些数据库还允许你创建自定义的数据类型别名，比如ip_address或者employee_id。在创建表的时候使用自定义的数据类型的好处是能让表结构更加清晰易读，另外也能让多个表中的字段定义更加一致。

之前的命令中的最后一个参数是可选的。如果这个参数被指定为NOT NULL，那么新的行数据（如果缺少相应的字段）的插入会被拒绝。在我们的例子中，主机名和 IP 地址对主机来说都是非常重要的，所以我们指定这两个字段是NOT NULL。所有其他的字段都是可选的（尽管它们也很有用），所以我们把它们设为NULL。在NULL/NOT、NULL之外还有一些约束，可以用来提高字段数据的一致性。比如我们可以用某种 SQL 方言来规定两台主机不能有相同的主机名，把下面的代码：

```
name      varchar(30)    NOT NULL,
```

改成：

```
name      varchar(30)    NOT NULL CONSTRAINT unique_name UNIQUE,
```

这里的 unique_name 就是相应的约束名。这样的命名约束可以使得违反约束而产生的错误信息更加可读。请参考你的数据库服务器说明文档来了解其他可以在表中使用的约束。

删除数据库上的一张表（比起创建来）则要简单得多：

```
USE sysadm
DROP TABLE hosts
```

向表中插入数据

现在我们有了一张空表，可以考虑如何加入新数据了。这时可以用两种方法，下面是第一种方法：

```
USE sysadm
INSERT hosts
VALUES (
    'shimmer',
    '192.168.1.11',
    'shim shimmy shimmydoodles',
    'David Davis',
    'Software',
    'Main',
    '309',
    'Sun',
    'Ultra60'
)
```

第一行告诉数据库服务器我们要处理的是sysadm中的数据。第二行开始我们选定hosts表进行插入操作，一次完成整行数据的插入。这个版本的INSERT语句是用来插入整行数据的（也就是说所有的字段都要插入）。^[注2]

如果只需要插入部分字段，可以声明需要填充的字段，好比下面的例子：

```
USE sysadm
INSERT hosts (name,ipaddr,owner)
VALUES (
    'bendir',
    '192.168.1.3',
    'Cindy Coltrane'
)
```

这个 INSERT 语句在遇到没有设定 NOT NULL 字段的时候会失败。

INSERT 还可以用来把一张表的数据插入另一张表，这在之后的例子中会介绍。在下面的例子中，我们都假定使用第一种 INSERT 格式来插入 hosts 表一行的所有字段。

注2： 有经验的 SQL 用户会建议你总是声明要插入的字段名，哪怕你要插入的是整行的数据。在这个例子中，我们虽然插入所有的字段，但还是可以考虑这个建议，因为它能让 INSERT 语句更少出错。由于省略字段名的语法往往依赖于字段的创建顺序，所以也有可能在修改表结构（比如新增字段）后出错。

查询信息

对于管理员来说，最常使用的 SQL 命令应该就是SELECT了。这个语句是用来从服务器查询信息的。在我们介绍这个语句之前，请注意：SELECT是SQL语言的灵魂所在。这里我们只能介绍它最简单的使用方法。关于如何书写更高效的 SQL（以及进行数据库的设计优化）这些更加深入的主题，最好参考其他的 SQL 和数据库专业书籍。

最简单的SELECT格式常常用来获取指定服务器和连接的信息。对于这类查询，不必指定数据源。下面三个例子就是这样：

```
-- Sybase/MS-SQL —— 获取服务器名称
SELECT @@SERVERNAME

-- MySQL —— 获取当前版本
SELECT VERSION();

-- Oracle —— 获取 STARTUP_TIME
SELECT STARTUP_TIME from v$instance;
```

这种获取数据库相关信息的查询对于不同的数据库来说差异非常显著。

获取一张表中所有行的信息

要获得 hosts 表中所有的数据，我们可以使用这样的 SQL 代码：

```
USE sysadm
SELECT * FROM hosts
```

这会返回所有的行和列的数据，列的顺序和表创建时的顺序一致：

name	ipaddr	aliases	owner	dept
bldg	room	manuf	model	
shimmer	192.168.1.11	shim shimmy shimmydoodles	David Davis	Software
Main	309	Sun M4000		
bendir	192.168.1.3	ben bendoodles	Cindy Coltrane	IT
West	143	Apple Mac Pro		
sander	192.168.1.55	sandy micky mickydo	Alex Rollins	IT
Main	1101	Dell Optiplex 740		
sulawesi	192.168.1.12	sula su-lee	Ellen Monk	Design
Main	1116	Apple Mac Pro		

如果我们需要获取某些字段，可以这样来指定它们：

```
USE sysadm
SELECT name,ipaddr FROM hosts
```

在指定了字段名的情况下，字段的输出则按照我们输入时的顺序，和表创建时的顺序没有关系。比如要查询每个大厦的 IP 地址，我们可以使用这个语句：

```
USE sysadm
SELECT bldg,ipaddr FROM hosts
```

这是它的返回信息：

bldg	ipaddr
-----	-----
Main	192.168.1.11
West	192.168.1.3
Main	192.168.1.55
Main	192.168.1.12

获取一张表中部分行的数据

如果你必须获取整个表的所有行，那么使用数据库会显得比较无聊。对于 SELECT 来说，我们可以加入 WHERE 子句来定义查询条件：

```
USE sysadm
SELECT * FROM hosts WHERE bldg='Main'
```

这会返回：

name	ipaddr	aliases	owner	dept	bldg
room	manuf	model			
-----	-----	-----	-----	-----	-----
shimmer	192.168.1.11	shim shimmy shimmydoodles	David Davis	Software	Main
309	Sun	M4000			
sander	192.168.1.55	sandy micky mickydoo	Alex Rollins	IT	Main
1101	Dell	Optiplex 740			
sulawesi	192.168.1.12	sula su-lee	Ellen Monk	Design	Main
1116	Apple	Mac Pro			

这个 WHERE 子句中可用的条件操作符合标准编程语言的模式：

= > >= < <= <>

与 Perl 不同的是，SQL 没有区分字符串和数字比较操作符。

条件操作符可以使用AND/OR来进行连接，也可以用NOT来进行反转。另外，我们还可以使用IS NULL来判定空字段，或者用IS NOT NULL来判定非空字段。比如下面的代码能够列出那些表中没有归属的主机：

```
USE sysadm
SELECT name FROM hosts WHERE owner IS NULL
```

如果你需要查询的是某个字段中的几个特定值，就可以使用IN操作符来指定值的列表：

```
USE sysadm
SELECT name FROM hosts WHERE dept IN ('IT', 'Software')
```

这会返回所有 IT 部门或者软件部门的主机。SQL 还提供了 BETWEEN 操作符来判定某个字段的值是否在某个范围中，这对于数字或者日期字段最实用。下面的例子显示了主楼的 10 层到 19 层的所有主机（假定房间编号是按照楼层设定的）：

```
USE sysadm
SELECT name FROM hosts
  WHERE (bldg = 'Main') AND
        (room BETWEEN '1000' AND '1999')
```

最后，在 WHERE 子句中还可以使用 LIKE 操作符来判定某个字段是否能匹配某个模式（不过这里的字符匹配比 Perl 的正则表达式要差不少）。比如下面的代码能列出所有别名中包含了“doodles”的主机：

```
USE sysadm
SELECT name FROM hosts WHERE aliases LIKE '%doodles%'
```

表D-2列出了所有 LIKE 操作符支持的通配符。

表D-2：LIKE 通配符

通配符	意义	最接近的等效 Perl 正则表达式
%	零个或者多个字符	.*
_	单个字符	.
[]	在某范围内的单个字符	[]

某些数据库服务器扩充了自己的 SQL 语法，加入了 SELECT 的正则表达式支持。比如 MySQL 就有 REGEXP 操作符，虽然这个操作符不能代替 Perl 的正则表达式，但是确实增强了标准 SQL 的匹配能力。

对查询结果的简单处理

最常用的三个SELECT语句的变化是COUNT、DISTINCT和ORDER BY。其中COUNT能对返回的行进行计数：

```
USE sysadm
SELECT COUNT(*) FROM hosts
```

而 DISTINCT 能够剔除结果中重复的记录。如果我们想列出所有主机的制造商，可以使用下面的语句：

```
USE sysadm
SELECT DISTINCT manuf FROM hosts
```

第三个有用的语句是 ORDER BY，它能指定返回结果的顺序。我们可以这样来指定返回的顺序：

```
USE sysadm
SELECT name,ipaddr,dept,owner FROM hosts ORDER BY dept
```

有经验的数据库用户常常习惯地在语句中加入 ORDER BY 子句，因为这样可以使得对结果的处理更加简单。

SQL 还有一些其他的操作符，它们能对查询结果做其他处理。比如说修改结果集的字段名，或者进行单字段/多字段的计算，或者修改字段的输出格式，或者进行子查询等等。请阅读专门的 SQL 书籍来了解 SELECT 语句的那些特殊语法。

把查询结果插入另一张表

对某些 SQL 服务器来说，可以通过 INTO 子句来将查询结果创建成一张表：

```
USE sysadm
SELECT name,ipaddr INTO itmachines FROM hosts WHERE dept = 'IT'
```

这条语句和之前的语句有相似的工作方式，只不过在最后，查询结果进入了一张叫做 itmachines 的新表。对于某些数据库来说，这个表如果不存在就会被自动创建。你可以把这个操作理解成大多数 Unix 和 Windows 平台上的 > 命令行输出转向。

注意：某些数据库（比如 MySQL^[注3]）不能支持 SELECT INTO 语句，它们往往需要用其他语法来实现这个功能，比如 Oracle 使用的是这样的语句：

```
CREATE TABLE COPY AS SELECT name,ipaddr FROM hosts WHERE dept = 'IT'
```

另外一些数据库需要使用 INSERT 语句来实现这个功能。对于 Microsoft SQL Server 和 Sybase 这样的数据库来说，数据库只要进行了相应的设定，就能使用这样的 SELECT INTO 语句。若没有相应的设定，这条语句会失败。

修改表中的数据

我们关于 SELECT 语句的很多知识也可以用在其他语句上。比如前面展示的 INSERT 语句也可以带上 SELECT 子句。这使得我们能将某个查询的结果数据插入至某张表。如果我们的

注3： 请特别注意，MySQL 5.x 版本引入了 SELECT .. INTO 语句，只不过它的输出是常规文件，而非这里介绍的数据库表。对于 MySQL 来说，对应的 SQL 语句是 INSERT .. INTO。

软件部门现在要合并到 IT 部门，我们可以把这两个部门的主机都合并到 `itmachines` 表中：

```
USE sysadm
INSERT itmachines
  SELECT name,ipaddr FROM hosts
  WHERE dept = 'Software'
```

如果我们需要修改表中的某一行数据，可以使用 `UPDATE` 语句来完成。如果公司的所有部门都搬迁到了某座叫做 `Central` 的大厦，那么可以这样修改所有行的大厦名称：

```
USE sysadm
UPDATE hosts
  SET bldg = 'Central'
```

大多数时候，我们只需要修改表中特定行的数据。为了完成这个任务，我们可以使用在 `SELECT` 部分介绍过的 `WHERE` 子句：

```
USE sysadm
UPDATE hosts
  SET dept = 'Development'
  WHERE dept = 'Software'
```

这段代码把软件部门改成了开发部门，下面的代码把 `bendir` 主机搬到了主楼：

```
USE sysadm
UPDATE hosts
  SET bldg = 'Main'
  WHERE name = 'bendir'
```

如果需要删除表中一行或多行数据（而非修改），我们可以使用 `DELETE` 语句：

```
USE sysadm
DELETE FROM hosts
  WHERE bldg = 'East'
```

不过目前还没有标准的方法来恢复 `DELETE` 造成的修改^[注4]，不过可以使用事务来获得某种程度的数据回滚（这里不多介绍）。大多数情况下，你应该先用 `SELECT` 语句来了解 `DELETE` 语句将要对数据造成的修改，然后再运行 `DELETE` 语句。不论如何，请特别注意这些修改操作的副作用。

注4：从 Oracle 10g 版起，增加了能够撤销 `DELETE` 和 `DROP` 操作的快速恢复功能，不过这项功能是有限的，要看删除数据的大小以及执行删除操作后数据库又有多少改动而定。

进行跨表的关联

关系数据库能提供多种跨表“融合”数据的功能，这个功能被称为跨表“连接”。考虑到判断条件的数量和开发人员的编码方式，连接可能会非常复杂。连接还有一些变化（比如内连接或者外连接），不过这里我们不打算深入介绍。如果你对这个感兴趣，请参考那些专门的 SQL 书籍。

下面就是一个连接的例子。这里我们需要引入另外一张名为 `contracts` 的表，其中含有每台主机维护人员的信息。这张表的结构如表D-3所示。

表D-3：我们的联系人表

名字	服务供应商	开始日期	结束日期
bendir	IBM	09-09-2005	06-01-2008
sander	Dell	03-14-2008	03-14-2009
shimmer	Sun	12-12-2008	12-12-2009
sulawesi	Apple	11-01-2005	11-01-2008

下面的代码能够连接 `hosts` 表和 `contracts` 表：

```
USE sysadm
SELECT contracts.name,servicevendor,enddate,bldg,room
FROM contracts, hosts
WHERE contracts.name = hosts.name
```

理解这段代码的一个方法是从中间开始读。`FROM contracts, hosts`告诉我们，服务器要连接的是`contracts`和`hosts`两张表。`WHERE contracts.name = hosts.name`的意思是对于`contracts`表的每一行都要找出`hosts`表中的相同`name`字段的行。注意，这里我们需要明确声明`contracts.name`，因为两个表都有`name`字段，所以这里需要明确用表名来区分它们。最后，`SELECT`这一行会产生我们需要的输出。

SQL 衍生

在我们结束这个教程之前，还有一些更加高级的 SQL 主题要介绍，这是为了让你能更好地解决一些其他问题。

视图

大多数 SQL 数据库都支持在表上创建视图。视图就好像固化的 `SELECT` 查询。一旦创建了包含某个 `SELECT` 查询的视图，相关的查询就被长久地记录下来。以后每次你通过视

图访问数据，原始的查询语句就会被调用。视图可以像常规表那样被查询。修改视图的数据往往会有限制，但如果服务器允许，相应的修改会发生在原始表上。

注意，这里不一定是发生在一张表上，有时候视图连接了多张表。所以视图背后的那些表可能都会被视图的修改而影响到。

创建视图时，还可以对某些字段进行计算，从而产生一个新的视图字段，这个功能就类似于电子表格的公式。另外还可以用视图来实现一些常规的任务，比如简化查询（例如，可以用来定制表中需要查询的字段）以及数据重构（例如，表用户看见的数据视图没有改变，即使底层表结构中的其他字段已被改变）。

下面的例子展示了视图的简化查询功能：

```
USE sysadm
CREATE VIEW ipaddr_view AS SELECT name, ipaddr FROM hosts
```

现在我们可以用非常简单的查询来返回需要的信息：

```
USE sysadm
SELECT * FROM ipaddr_view
```

这个查询的输出是：

name	ipaddr
shimmer	192.168.1.11
bendir	192.168.1.3
sander	192.168.1.55
sulawesi	192.168.1.12

视图的删除也是通过 DROP 语句来完成的，类似表的删除：

```
USE sysadm
DROP VIEW ipaddr_view
```

删除视图对于它背后的表并没有影响。

游标

到目前为止，我们所有的查询都是直接从服务器返回所有的值。但是有时候，逐行返回查询结果也很有意义。往往在大查询中嵌入其他 SQL 的时候会用到游标。如果你的查询结果超过了千行，往往逐行处理更加有效，因为这可以避免把所有的数据结果一股脑地放入内存。Perl 的 SQL 编程往往需要和游标打交道。下面的服务器端 SQL 代码展示了 Sybase 或者 Microsoft SQL Server 中游标的使用：

```

USE sysadm
-- 声明我们自己的变量
DECLARE @hostname varchar(30)
DECLARE @ip varchar(15)

-- 声明我们自己的游标
DECLARE hosts_curs CURSOR FOR SELECT name,ipaddr FROM hosts

-- 打开该游标
OPEN hosts_curs

-- 逐个遍历每张表，每次取出整行内容，
-- 直到收到错误为止
FETCH hosts_curs INTO @hostname,@ip
WHILE (@@fetch_status = 0)
    BEGIN
        PRINT "----"
        PRINT @hostname
        PRINT @ip
        FETCH hosts_curs INTO @hostname,@ip
    END

-- 关闭游标（严格来说如果后续使用 DEALLOCATE 命令，则此处就并非必须）
CLOSE hosts_curs

-- 销毁游标
DEALLOCATE hosts_curs

```

这会产生下面的输出：

```

----
shimmer
192.168.1.11
----
bendir
192.168.1.3
----
sander
192.168.1.55
----
sulawesi
192.168.1.12

```

存储过程

大多数数据库系统允许我们把SQL代码存储在服务器端，在那里服务器对它进行编译和优化，从而获得更快的执行速度。这种上传的代码往往称为存储过程（stored procedure）。这些存储过程对管理员来说非常关键，因为大多数数据库系统的维护必须使用它们。比如要修改 `sysadm` 数据库的所有者，就必须这样做：


```
USE sysadm  
sp_changedbowner "jay"
```

有些数据库还支持称为“触发器”（trigger）的程序。触发器是那些在特定事件发生的时候（比如一行被 INSERT 的时候）能自动执行的存储过程。每种数据库的触发器实现都不太相同，所以请参考你所使用的数据库的说明文档来了解如何执行 CREATE TRIGGER 和 DROP TRIGGER 语句。

相信你现在已经获得了必要的 SQL 知识，是时候开始阅读第 7 章。

5分钟RCS教程

这个简明教程会告诉你如何使用 RCS (Revision Control System, 修订控制系统) 来进行系统管理。RCS 适合用来对所有的系统文件进行版本控制。它的功能比我们介绍的还要多, 所以想认真使用的时候请查看手册页 (以及本附录末尾列出的参考资料)。你也可能会好奇, 为什么我们会介绍 RCS, 毕竟现在那些时尚的版本控制系统 (比如 Git 和 Subversion) 都在大行其道。这是个很好的问题, 我们会在这个教程的后面解答。眼下我们还是先开始介绍 RCS 的基础概念, 这会有助于理解后面的解释。

RCS 的工作原理类似于汽车租赁公司。同一时间只有一个人能租一辆车, 租赁公司也只能出租那些登记好的车辆。客户可以随时查看车辆列表, 但是如果两个人都想租同一辆车, 那么后面的人必须等前面那个人还车后才行。最后, 在客户归还车辆的时候, 租赁公司会小心检查, 记录车辆有什么变化。所有这些对 RCS 来说也一样。

在 RCS 中, 文件就像是汽车。如果你想使用它来跟踪某个文件, 那么必须在开始之前把它 “check in”。

```
$ ci -u inetd.conf
```

ci 意味着 “check in”, 而 -u 的意思是在 check in 期间不要动 *inetd.conf* 文件。当一个文件在 check in (也就是说预备出租) 的时候, RCS 要做两件事情来提醒用户文件在它的控制下:

1. 删除原始文件, 只留下文件的 RCS 归档。这有时候会让初学者困惑, 因为文件一旦 check in 就消失了, 但其实文件是被藏到 RCS 归档文件里面去了。这个归档文件常常被命名为 *filename,v*, 它的位置有时候在原始文件的目录下面, 有时候在一个名为 RCS 的子目录里面 (往往是用户创建的)。你必须像对待原始文件那样妥善设置 RCS 目录和归档文件的系统读写权限。

2. 如果像范例里面那样使用了 `-u` 开关, 那么会立即把文件 check out, 这会导致文件读写属性被设置为“只读”。

要修改某个受到 RCS 保护的文件 (就像租车之前), 你必须先对那个文件做 check out (`co`):

```
$ co -l services
```

这里的 `-l` 参数告诉 RCS 把文件“严严实实地”锁住, 这意味着其他企图 check out 的用户会被拒绝。这个锁只是对 RCS 而言的, 并不是任何意义上的文件系统锁 (不会修改 ACL、文件属性等等)。其他在 `co` 命令时候常用的开关有:

- `-r <revision number>` 用来 check out 文件的某个老版本
- `-p` 用来“输出”一个过去的版本到屏幕, 但会跳过真正的 check out

在修改文件之后, 你需要使用一条老命令来把它放回 RCS 的控制之下 (`ci -u filename`)。这次的 check in 操作会比较聪明地存储新文件里的变动, 以节省空间。

每次被修改并 check in 的文件都会有一个新的版本号。在 check in 的时候, RCS 提示你输入的那些注释也会被自动记录下来。这些历史信息 (以及当前 check out 留下的信息) 可以使用 `rlog filename` 命令查看。

如果某人在 check out 之后拒绝把新文件重新 check in 回 RCS 系统 (可能那天生病在家) 而你又急着需要对文件做自己的修改, 那么可以使用 `rcs -u filename` 命令强行打开那个人留下的锁。这个命令会要求你输入一条信息, 并自动发邮件给那个人。

在强行开锁以后, 你可能想要检查文件的当前版本和 RCS 归档的差异。`rcsdiff filename` 命令能够告诉你这个差异。如果你认可这些修改, 那么可以 check in 这个文件, 并且可以立刻再 check out 这个文件以便进一步修改。类似于 `co` 命令, `rcsdiff` 也可以带上 `-r <revision number>` 标志来比较两个归档版本之间的差异。

表E-1列出了一些常规的 RCS 操作和相应的命令。

表E-1: 常规 RCS 操作

RCS 操作	命令
文件的初始 check in (保持文件在文件系统中可用)	<code>ci -u filename</code>
带锁 check out	<code>co -l filename</code>
check in 并解锁 (保持文件在文件系统中可用)	<code>ci -u filename</code>

表E-1：常规 RCS 操作（续）

RCS 操作	命令
显示某个文件的 <i>x.y</i> 版本	<code>co -p x.y filename</code>
撤回至 <i>x.y</i> 版本（用历史版本覆盖文件系统中的文件归档）	<code>co -r x.y filename</code>
比较文件系统版本和最近归档的差异	<code>rcsdiff filename</code>
比较 <i>x.y</i> 和 <i>x.z</i> 版本的差异	<code>rcsdiff -r x.y -rx.z filename</code>
检查 check in 记录	<code>rlog filename</code>
强行打开其他人在某个文件上设置的 RCS 锁	<code>rcs -u filename</code>

不论你相信与否，这就是你需要了解的所有 RCS 基础。一旦你开始使用它进行系统管理，你会发现这次的学习非常值得。

选择RCS（而不是CVS、Git、SVN等等）的理由

既然有那么多比 RCS 更新更酷的版本控制系统，为什么我还要建议你学习 RCS 呢？其实我在某些时候也会使用那些新的系统，但是与 RCS 相比，它们并不适合在系统管理中使用。

对 CVS 和 SVN 来说，用户往往从中央归档库中 check out 某个感兴趣的文件到某个“工作目录”。多人可以同时 check out 同一个文件，如果他们的修改并不冲突，那么系统会在 check in 时尝试合并（融合）所有的改变。Subversion 说明文档里面把这种工作模式称为“拷贝—修改—合并”。

有几个问题使得这种工作模式在系统管理的领域（对配置文件来说）不太合适：

- 配置文件在文件系统的位置至关重要。如果 `/etc/passwd` 文件不在 `/etc` 目录下，或者被版本控制系统修改成特殊的格式，那么它就会完全失效。确实有可能强迫每个人都使用 `/etc` 作为工作目录，但这往往会导致更多危险。RCS 则能在 check in 之后直接在文件系统原来的位置使用文件。
- 拷贝—修改—合并的工作模式和系统管理的模式不吻合。多人可以同时编辑某个配置文件，并且由系统自动完美融合（不会发生段落之间的相互冲突），这种想法确实太理想化了。虽然 CVS 和 SVN 都能支持锁住文件并拒绝他人修改的模式，但是这与它们本身的设计思想相悖。相比之下，RCS 的锁机制则简单有效。如果确实需要对数据进行并发修改，那么我建议你使用数据库，配合自动生成配置文件的脚本，那应该是更好的解决方案。

- CVS 和 SVN 是“基于目录的”：它们擅长处理目录树中的文件。Git 是“基于内容的”，它管理的是数据。RCS 则是“基于文件的”。
- 尽管那些新的版本控制系统往往都支持离线操作，但是使用一个中心网络化归档文件库来管理所有的重要文件常常会导致很多问题。如果你的机器因为某种问题而无法上网，而这个时候你还有 RCS 归档库和 RCS 可执行程序，那么你就很容易完成修复。但是如果你的版本控制系统需要你联线来修复网络问题，那么这时候问题就会挂在那里。Git 虽然有很好的全分布式管理模型，但是在这种情况下它也会有自己特殊的问题。
- 简而言之，尽管 CVS、SVN、Git 或其他的版本控制系统可能有更强的功能，但是那些功能往往是为软件开发而设计的，在管理系统文件的时候 RCS 更加适合实际需要。

更多参考信息

<ftp://ftp.gnu.org/pub/gnu/rcs> 有 RCS 软件包最新的源代码（如果你的操作系统没有自带，大部分标准包管理器也都包含了该软件包）。

<http://cygwin.com> 是 RCS 软件包的来源之一（以及很多其他 Unix 下的程序）。如果你想在不安装整个 Cygwin 环境的前提下安装 RCS，可以使用这个版本：<http://www.cs.purdue.edu/homes/trinkle/RCS>。

《Applying RCS and SCCS: From Source Control to Project Control》，由 Don Bolinger 和 Tan Bronson 所著（O'Reilly 出版），是很棒的 RCS 参考书。

如果你在 RCS 中找不到所需的某种功能或特性，可以查看 <http://www.nongnu.org/cvs> 和 <http://subversion.tigris.org>，它们即是你下一步需要了解的并行版本系统（CVS）和子版本（Subversion, SVN）。

在 CVS 和 SVN 之后，还有一些较新的分布式版本控制系统，如 *git*、*mercurial*、*bazaar* 和 *darcs*。如果需要了解更多，请查看 http://en.wikipedia.org/wiki/Distributed_Version_Control_System 上的这篇文章。

2分钟VBScript翻译到 Perl教程

怎么会有一本关于 Perl 的书居然还介绍 VBScript 呢？在你开始本能地拒绝这个附录之前，我要先向你解释为何有必要花两分钟时间来学点 VBScript。下面的所有介绍都假定你对基于 Windows 的操作系统有些使用经验。如果你从来没有用过 Windows，并且将来也不计划接触它，那么请跳过这个附录，其余的人可以留下来听我介绍。

这可能会让某些人觉得有些饶舌，不过微软确实希望管理员用微软的技术来将管理任务自动化。Perl 之所以能搭上这班车多亏了 Jan Dubois 和其他人在 Win32::OLE 上投入的精力。这个模块给 Perl 程序带来了几乎与 VBScript 脚本同样的与微软系统交互的能力。

要知道的是，Win32::OLE 虽然使得这种交互成为可能，但是并不总是最简便的方法。Perl 并没有 VBScript 语言的 DWMM (Do What Microsoft Means) 特质，所以这意味着某些看似简单的 VBScript 代码并不能很容易地翻译成 Perl 脚本。另外，这个问题还因为缺乏参考资料和技术文档而显得更加突出。除了像 David Roth 的书这样的典型例外，绝大多数 Windows 脚本编程方面的范例都是使用 VBScript 来实现的。比如微软那卓越的 Script Center (脚本中心) 网站 (它基于同样高质量的 Windows 2000 Scripting Guide) 就是最好的参考资料，只不过里面的范例全部是用 VBScript 编写。

我并不是一名 VBScript 开发人员，也并不打算让你成为这样的人。所以即使读过这个附录之后你大概也不能摇身一变成为这方面的高手。好在你不必懂得非常多的 VBScript (或者有深入的 Win32 编程经验) 才能把 VBScripts 脚本翻译成 Perl 的 Win32::OLE 脚本。这个附录会展示一些基本的翻译案例，说明如何实际操作，其中有些脚本就来自微软的脚本中心。

翻译策略

作为首先展示四个翻译策略之一，我们先尝试翻译下面的简单 VBScript：

```
' 列出 fabrikam.com 管理团队的所有成员

Set objGroup = GetObject _
("LDAP://cn=managers,ou=management,dc=fabrikam,dc=com")
For each objMember in objGroup.Members
Wscript.Echo objMember.Name
Next
```

我们稍后还会介绍其他几个范例脚本，这里我们先尝试翻译这段比较简单的代码。^[注1]

策略一：载入模块

所有翻译的程序都应该在开始前先载入 Win32::OLE 模块：

```
use Win32::OLE;
```

如果你认为脚本需要用到容器和相应的容器对象^[注2]，你应该同时导入in原语或者载入 Win32::OLE::Enum 模块：

```
# 'in' 是另一种表述 Win32::OLE::Enum->All() 的方法
use Win32::OLE qw(in);
# 或者用
use Win32::OLE;
use Win32::OLE::Enum;
```

另外还可以载入 Win32::OLE::Const 模块，因为它可以导入那些应用程序或者操作系统库级别的常量。稍后你会看到它的应用。

另外还有一些原语你可能也需要导入，比如 with 和 valof，它们可以用来翻译那些更加复杂的脚本。不过使用它们通常需要更加深入的 Windows 编程方面的知识。请参考 Win32::OLE 的说明文档来了解它们的使用方法。

策略二：创建对象的引用

这里的翻译非常直白：

```
my $objGroup =
```

注1：我不喜欢 VBScript 的一个原因是它需要特殊的续行字符（这里用的是 _）来把两行代码表示为一行代码。但是没办法，谁让我们需要理解另外一种语言呢？

注2：请参考第9章的“处理容器对象/集合对象”一节来了解更多的容器对象。

```
Win32::OLE->
    GetObject('LDAP://cn=managers,ou=management,dc=fabrikam,dc=com');
```

为了使 VBScript 和 Perl 脚本更加相似，我们这里保留了 VBScript 的变量名，延续了大小写混杂的格式。

策略三：使用哈希解引用语法来访问对象属性

VBScript 使用点号 (.) 字符来访问对象的属性。相应的 Perl 代码^[注3]则需要使用哈希解引用语法（也就是 `$objGroup->{property}`）。

所以，下面的 VBScript 代码：

```
objGroup.Members
```

会被翻译成这样的 Perl 代码：

```
$objGroup->{Members}
```

策略四：处理容器对象

无论是上一节的原始 VBScript 还是翻译过的 Perl 代码都会返回容器对象。这个对象中含有一系列用户对象（也就是那些 *managers* 组的成员用户）。VBScript 能用奇怪的 `in` 语句来获取容器中的对象，不过现在我们可以使用这个语句（这要感谢策略一导入的函数）：

```
for my $objMember (in $objGroup->{Members}){
    # 使用策略三中介绍的访问语法
    print $objMember->{Name}, "\n";
}
```

现在你完成了你的第一个从 VBScript 翻译出来的 Perl 程序：

```
# 列出 fabrikam.com 管理团队的所有成员

use Win32::OLE qw(in);

my $objGroup =
    Win32::OLE->
        GetObject('LDAP://cn=managers,ou=management,dc=fabrikam,dc=com');

for my $objMember (in $objGroup->{Members}){
    print $objMember->{Name}, "\n";
}
```

注3： 这里请注意，我们针对的是 Perl 第五版。对于正在开发中的 Perl 第六版来说，也会使用这个点号字符。

这是不错的兆头，因为代码看上去还挺简单的。要知道，这里的目标就是能对简单的系统管理 VBScript 脚本进行 Perl 的翻译，而不必涉及太过复杂的领域。如果你觉得有点复杂，没关系，相信进行这种翻译的次数越多，你就会越熟练。下面我们换一个来自微软脚本中心的代码，以便介绍另外一个翻译策略。

策略五：翻译方法调用

```
' 创建一个新的全球安全组 -- atl-users02 -- 使用活动目录。
```

```
Set objOU = GetObject("LDAP://OU=management,dc=fabrikam,dc=com")
Set objGroup = objOU.Create("Group", "cn=atl-users02")
objGroup.Put "sAMAccountName", "atl-users02"
objGroup.SetInfo
```

第一行代码应该不难翻译，所以我们先看其他几行。在这里，点号字符 (.) 的用处和之前在策略三中看到的不太相同，这里不再是用来访问对象的属性（也就是数据），而是调用它的方法（也就是程序）。仿佛命运的安排，Perl 的方法调用语法类似于哈希解引用。Perl 使用箭头操作符 (->) 来表达两种操作，所以上面几行的代码可以翻译成：

```
my $objGroup = $objOU->Create('Group', 'cn=atl-users02');
$objGroup->Put('sAMAccountName', 'atl-users02')
$objGroup->SetInfo;
```

下面就是翻译后的结果：^[注4]

```
# 创建一个新的全球安全组 -- atl-users02 -- 使用活动目录。

use Win32::OLE;

my $objOU = Win32::OLE->
    GetObject('LDAP://OU=management,dc=fabrikam,dc=com');

my $objGroup = $objOU->Create('Group', 'cn=atl-users02');

$objGroup->Put('sAMAccountName', 'atl-users02')

$objGroup->SetInfo;
```

很简单，不是吗？不过这里还有个比较模糊的地方，那就是最后一行 `SetInfo` 的翻译。这里我们该如何知道它是方法调用还是属性访问呢？当然这里的代码没有赋值操作，所以很有可能是方法调用，因为属性访问之后最常见的操作就是对返回值的处理。这里原始的 VBScript 看上去并不打算使用返回结果，所以我们可以大胆推测这是一个方

注4： 这里省略了一些错误处理的逻辑，因为这里关注的是如何进行跨语言的翻译。但是无论如何，错误处理总是必要的，比如这里可以检查 `Win32::OLE::LastError()` 的返回值。

法调用。另外一个小窍门就是因为“SetInfo”这个词看上去就是一个动作，而不像是数据的名字。如果它看上去像是动作的话，理应是一个方法调用，而不是一个属性容器。不过这个窍门并非万全的，有时候只是帮你找到线索而已。实在不能推测的情况下，可以先把它翻译成一个属性访问，失败之后，再尝试方法调用。

策略六：处理常量

现在让我们介绍最后一个与翻译策略相关的 VBScript 范例。

```
' 从 Sea-Users 组中去除用户 MyerKen。

Const ADS_PROPERTY_DELETE = 4

Set objGroup = GetObject _
    ("LDAP://cn=Sea-Users,cn=Users,dc=NA,dc=fabrikam,dc=com")

objGroup.PutEx ADS_PROPERTY_DELETE, _
    "member", _
    Array("cn=MyerKen,ou=Management,dc=NA,dc=fabrikam,dc=com")

objGroup.SetInfo
```

这段代码的第一行可能会让你眼前一亮。在 VBScript 中，Const 是用来定义常量的。这些常量是由操作系统或者应用程序开发者定义的，存储在某个组件或应用的类型库中。VBScript 的一个限制在于它不能读取这些库中定义的常量（比如 ADS_PROPERTY_DELETE 这样的常量），必须由开发人员在脚本中自行定义。而 Perl 因为有了 Win32::OLE::Const 这样的模块，就不必受到这个限制。所以我们不必对这行进行翻译，可以转而这样写：

```
use Win32::OLE::Const 'Active DS Type Library';
```

然后 ADSI 常量就可以使用了。下一个比较常见的问题就是从哪里可以查到“Active DS Type Library”这样的魔力字符串呢？为什么它不是“ADSI TypeLib”或者“ADS Constants Found Here？”这个字符串的来源是 Windows 注册表中的 *activeds.tlb* 文件，参考位置是 *HKCR\TypeLib* 或者 *HKLM\Software\classes\TypeLib*。如果这对你来说还不够明显，那么还可以尝试以下方法：搜索注册表、阅读微软的 SDK（或者其他说明文档），另外还可以在网络上搜索其他人的范例代码，直到最终找到这个字符串为止。

第二行和第四行的代码之前已经看过了，所以我们把重点放在第三行上。之前已经看过如何执行方法调用，现在又知道了如何导入常量，剩下的问题就是如何处理 `Array("cn=MyerKen...")` 了。好消息是 VBScript 的 `Array()` 创建等效于 Perl 的匿名数组引用创建语法：

```
$objGroup->PutEx(ADS_PROPERTY_DELETE,  
    'member',  
    ['cn=MyerKen,ou=Management,dc=NA,dc=fabrikam,dc=com']);
```

下面是我们最终翻译出来的代码：

```
# 从 Sea-Users 组中去除用户 MyerKen。  
  
use Win32::OLE::Const 'Active DS Type Library';  
  
my $objGroup = Win32::OLE->  
    GetObject('LDAP://cn=Sea-Users,cn=Users,dc=NA,dc=fabrikam,dc=com');  
  
$objGroup->PutEx(ADS_PROPERTY_DELETE,  
    'member',  
    ['cn=MyerKen,ou=Management,dc=NA,dc=fabrikam,dc=com']);  
  
$objGroup->SetInfo;
```

相信这六个翻译策略能帮你解决不少脚本转化问题。

更多参考资料

如果你还没有安装微软的 Scriptomatic 工具（目前已经是第二版），那么请从 <http://www.microsoft.com/technet/scriptcenter/tools/scripto2.msp> 下载它。这个来自“Microsoft Scripting Guys”的工具能让你浏览本机的 WMI 名称空间。一旦你发现任何感兴趣的东西，它就能帮你生成相应的脚本，是不是很棒？不过更棒的是它能生成的脚本使用的脚本语言包括 VBScript、JScript、Perl 和 Python。我实在找不到跨语言脚本翻译方面更好的对比工具了，这是为什么我总是在 WMI 相关的章节里推荐它。另外，如果你希望在 Vista 下使用这个工具，请先参考第 1 章关于 Vista 的章节。

最后我还得提到的是商业化的产品，因为你也许懒得自己动手翻译 VBScript，另外也许你不介意采购一个更加复杂的软件包来完成这样的简单任务。VBScript Converter 是 ActiveState 的 Perl Dev Kit (PDK) 软件包的成员。关于此产品的更多信息可以参考 http://activestate.com/perl_dev_kit/。

20分钟SNMP教程

简单网络管理协议（Simple Network Management Protocol, SNMP）是广泛使用的网络设备管理协议。不过如同在第12章开头提到的那样，SNMP 并非名副其实的“简单”。这个“简明”教程就是为了让让你获得足够多的关于 SNMP 第一版的信息而引入的。

SNMP 的作用是让管理终端能够通过远程设备上运行的SNMP 代理查询信息。在重要情况发生的时候（比如计数器超过阈值），代理也能主动给管理终端发信息。当我们在第12章使用 Perl 进行 SNMP 编程的时候，我们其实是在扮演管理终端的角色，对远程设备上运行的 SNMP 代理进行查询。

我们在这个教程中关注的是 SNMP 的第一版。尽管目前 SNMP 已经有了七个版本的协议提案（SNMPv1、SNMPsec、SNMPv2p、SNMPv2c、SNMPv2u、SNMPv2* 和 SNMPv3），但是第一版仍然是使用最广泛的，当然因为安全性的提升，第三版也有很好的应用前景。

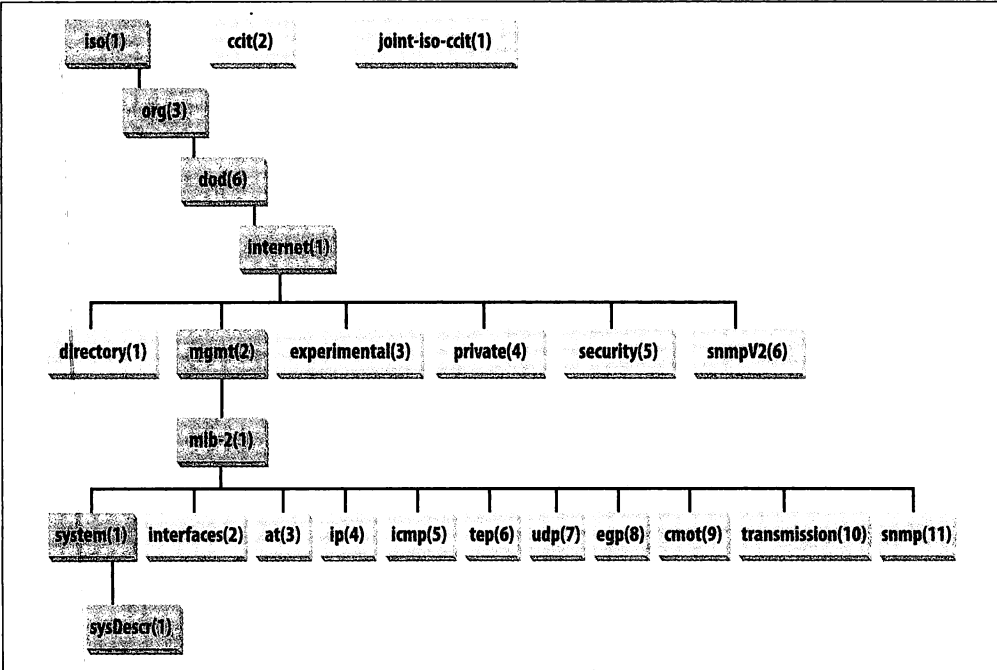
Perl 和 SNMP 都支持简单数据类型。Perl 使用变量作为最基础的数据类型，而列表和哈希可以说是标量的集合体。在 SNMP 中，也可以使用标量“变量”。SNMP 变量可以存放四种类型的数据：整数、字符串、对象标识符（稍后详细介绍）以及空值，而且在 SNMP 中，一系列相关的变量也可以组合成更大型的数据结构（主要是表）。以上就是两者主要的相似之处。

Perl 和 SNMP 在变量名方面开始显示出差异。对 Perl 来说，虽然有稍许的限制，但是你几乎可以随意选择变量名。而 SNMP 变量命名则受到很多限制。所有的 SNMP 变量都在一个叫做管理信息库（Management Information Base, MIB）的虚拟层次结构中。所有的变量名都必须在此框架中定义。MIB（包括现在的MIB-II版）规定了所有的可管理对象（及其名字）都是以树形结构组织的。

在树形结构方面，MIB 非常类似于文件系统，只不过 MIB 中存放的是管理信息，而不

是文件。树中的每个对象都有一个短的文本字符串，称为标签（label），以及代表节点在树中位置的相应数字。为了让你看到这个系统的组织结构，让我们查看 MIB 中存放系统描述信息的 SNMP 变量。请紧随我的脚步，我们要在树中走八层才能到达那里。

图G-1显示了一棵自顶向下的 MIB 树。



图G-1：在 MIB 中寻找 sysDescr(1)

树的顶层有三个标准的组织：iso(1)、ccitt(2)和joint-iso-ccitt(3)。在iso(1)节点下面有一个叫做 rg(3) 的节点用来容其他组织。在这个节点的下面有有个名为dod(6)的节点，代表Department of Defense。在那个节点下面有一个叫做internet(1)的节点，代表互联网社区。

现在开始名字显得越来越有意义了。internet(1) 这个子树的内容是由互联网架构委员会(Internet Architechture Board)指定的，具体内容请参考表G-1。

表G-1：internet(1) 节点的子树

子树	描述
directory(1)	OSI 目录
mgmt(2)	RFC 标准对象
experimental(3)	互联网实验

表G-1: internet(1) 节点的子树 (续)

子树	描述
private(4)	指定供应商
security(5)	安全
snmpV2(6)	SNMP 内部

因为我们感兴趣的是使用 SNMP 来管理设备，所以会进入 `mgmt(2)` 分支。这个节点下面的第一个节点是 MIB 本身，这有点递归的意思。因为它是这个子树的唯一的节点，所以它的名字是 `mib-2(1)`。

在这里我们会发现 MIB 真正有趣的东西。这是我们打开的所有子树中第一个看上去真正有用的，因为几乎其中所有的变量都有值得查询的信息：

```
system(1)
interfaces(2)
at(3)
ip(4)
icmp(5)
tcp(6)
udp(7)
egp(8)
cmot(9)
transmission(10)
snmp(11)
```

记住我们要寻找的是存储系统描述信息的 SNMP 变量，所以 `system(1)` 组看上去更加合乎需求。这棵子树的第一个节点就是 `sysDescr(1)`。哈哈，我们终于找到了需要的对象。

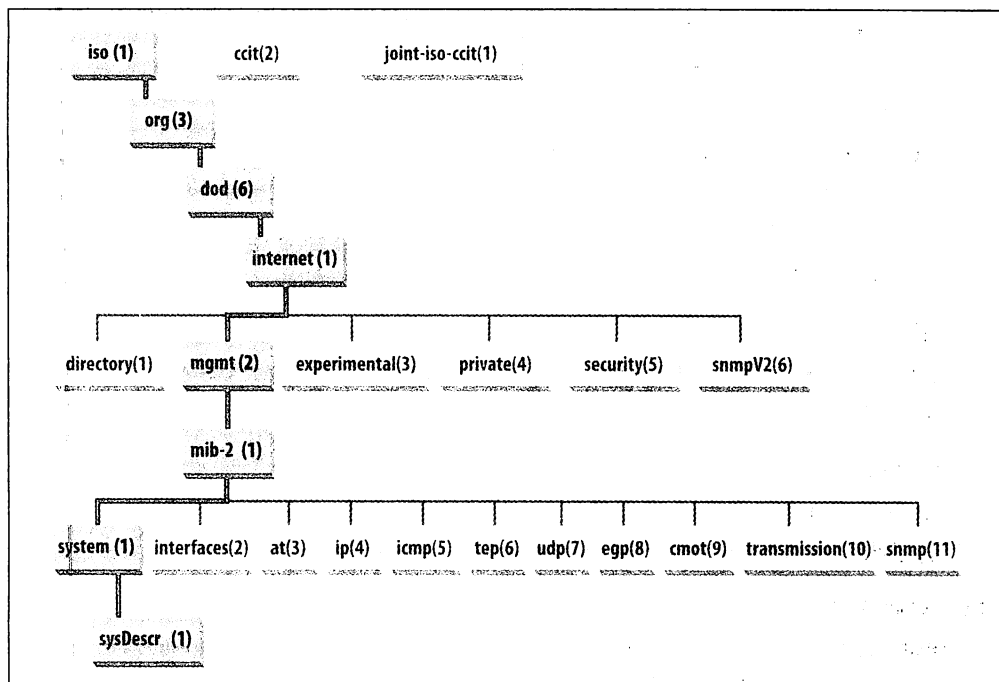
为什么要花这么大力气在树里面找来找去呢？这是为了让我们了解 `sysDescr(1)` 的对象标识符（Object Identifier, OID），也就是在路上遇到的所有对象编号的点分格式。图 G-2 展示了这条路径。

这样看来互联网的 OID 是 `1.3.6.1`，而系统对象组的 OID 是 `1.3.6.1.2.1.1`，`sysDescr` 对象的 OID 则是 `1.3.6.1.2.1.1.1`。

在我们实际使用 OID 的时候，还需要加上另外一个数字才能获得这个变量的值。在这里我们需要在数字字符串的末尾再加上 `.0` 来表示这是对象的第一个实例（其实一个设备也只能有一个描述信息）。

我们现在就可以来看看这个变量的值。在附录中我们会使用 Net-SNMP 软件包提供的命令行工具来进行演示。这个软件包 (<http://www.net-snmp.org>) 是绝佳的 SNMP 第一版和

第三版的自由软件实现。我们介绍这个工具是因为有个 Perl 模块也在使用它提供的库程序，而且其他工具的使用方法也大致类似。一旦你熟悉了 SNMP 的命令行工具，那么使用 Perl 来实现也会非常容易。



图G-2: 为我们需要的对象找出 OID

Net-SNMP 命令行工具允许我们在 OID 或变量名之前加上一个点号 (.) 来表明这个对象的起点是树的顶端。这里我们有两种方法来查询 *solarisbox* 主机的系统描述信息（注意第二个命令其实本来应该写在一行中的，这里断行是为了提高可读性）：

```
$ snmpget -v 1 -c public solarisbox .1.3.6.1.2.1.1.1.0
$ snmpget -v 1 -c public solarisbox \
    .iso
    .org.dod.internet.mgmt.mib-2.system.sysDescr.0
```

这两个命令会得到同样的结果：

```
system.sysDescr.0 = Sun SNMP Agent, Ultra-1
```

现在回到理论部分。请注意 SNMP 这个缩写里面的 P 代表的是 Protocol。SNMP 本身是用来在管理者和被管理者之间传送信息的协议。这个协议传送的数据对象，或者称为协

议书数据单元 (Protocol Data Unit, PDU)，是非常简单的。以下就是常见的 PDU，尤其是在通过 Perl 来访问时是最常用的：^[注1]

get-request

get-request 是其他 PDU 的兄长，用来获取 SNMP 变量的值。很多和 SNMP 打交道的人只知道如何用它就足以完成任务了。

get-next-request

get-next-request 类似于 **get-request**，区别在于它返回的是指定对象之后的对象（也就是 RFC 中所说的“第一个字典顺序的后继者”）。这个操作用来查找某个逻辑表中的所有元素。比如你可以使用连续的 **get-next-request** 操作来获取某台主机的 ARP 表的每一行。稍后我们会看到它的实际应用。

get-bulk-request

get-bulk-request 是 SNMP 第二版或第三版加入的操作，允许批量传送信息。**get-bulk** 能让你一次查询一组值，所以是更高效的信息传送方式。

set-request

set-request 操作的效果名副其实，它用来修改 SNMP 变量的值。这个操作主要是为了修改 SNMP 设备的配置信息而设立的。

trap/snmpV2-trap

trap 是 SNMP 第一版中的名字，在第二和第三版中它的名字是 **snmpV2-trap**。它能让你要求 SNMP 设备主动报告某些事件（比如重启或者计数器达到阈值），而不必反复查询。这也使得事件在第一时间得到处理，不必被动等待下一次通过代理查询。

inform-request

inform-request 是在 SNMP 第二版和第三版加入的 PDU，它提供了与 **trap** 类似的功能，还带上确认机制。对于普通的 **trap** 信息来说，代理虽然可以发送此信息，但是无从知道管理终端是否收到，而这个新的 PDU 就有确认机制。

response

response 是用来对任何 PDU 进行响应的 PDU。比如它可以用来答复 **get-request**，也可以用来说明 **set-request** 是否成功。在 SNMP 编程的时候，你完全可以忽略这种 PDU，因为大多数 SNMP 库程序都会自动处理它。不过，知道请求机制是如何答复的也有好处。

注1：SNMP 第二版和第三版中包含的 PDU 是通过 RFC 3416 定义的，它是建立在 SNMP 第一版的 RFC 1157 基础上的。不过 RFC 中也没有包含太多的其他 PDU，所以这里介绍的基本上够用了。

如果你是第一次和 SNMP 打交道，可能会很自然地问：“这就是全部了吗？get 加上 set，以及事件通知，这就是 SNMP 的功能了？”先别着急，从字面上来说这个模型虽然“简单”，但是并不意味着缺乏功能。只要 SNMP 设备的制造商对变量进行认真设计，那么使用这个协议就能完成所有任务。RFC 中典型的例子就是对 SNMP 设备进行重启。虽然协议中没有设计“reboot-request”这样的 PDU，不过设备制造商可以通过一个 SNMP 变量来设置重启倒计时的秒数。一旦这个变量通过 set-request 来设置，触发器就会让这个设备在若干秒之后重启。

有了这种控制能力，一个自然的问题就是如何避免大家随意重启机器呢？应该说，这个问题的解答在协议的早期显得非常稚嫩。所以有人戏称这个协议为“Security Not My Problem”（其缩写同样也是 SNMP），这要归功于 SNMP 第一版的验证机制。要了解安全问题中的谁、做什么和怎么做，我们需要引入一些新名词，所以请耐心看下去。

SNMP 第一和第二版允许你通过一种叫做 *communities*（团体）的机制来定义 SNMP 协议参与者之间的关系。团体可以用来对相似访问权限的 SNMP 代理以及访问终端进行分组。所有具有同样团体名称的终端都有同样的访问权限，而证明自己属于这个团体的方法就是看你知不知道这个团体的名称。这个方案就是对安全问题中的“谁”的解答。

现在是关于“做什么”部分。RFC 1157 把 MIB 对部分管理终端可访问的设置叫做 SNMP MIB view。比如很自然地，支持 SNMP 的电烤箱^[注2]应该不会和 SNMP 路由器提供同样的 SNMP 配置变量。

MIB 中的每个对象的访问属性都是 read-only、read-write 或者 none 其中之一。这也可以成为对象的 SNMP 访问模式。如果我们把 SNMP MIB view 和 SNMP 访问模式结合起来，就能定义一个 *SNMP community profile*，也就是具体哪个团体能够访问何等属性的哪些 MIB 变量。

一旦我们把“谁”和“做什么”关联起来，就能产生 SNMP 访问策略。它定义了某个团体的成员可以访问的具体变量范围。

现在是问题的“怎么做”的部分，如何把这些关联起来呢？日常应用中，你可以把路由器或者主机放在至少两个团体里面，一个团体只能读取，而另一个团体可以读写。通常把只读团体称为 *public*，而另一个称为 *private*，而这也是最常见的默认团体名。比如对于思科路由器来说，你可以做这样的配置：

！设置只读团体名为 MyPublicCommunityName

注2： Web上确实曾经有过几台支持 SNMP 的可乐贩卖机，所以电烤箱也是有可能的。要是你觉得好笑的话，我得告诉你世界上第一台互联网烤箱（还能通过 SLIP 支持 SNMP 协议）早在 1990 年就被造出来了。

```
snmp-server community MyPublicCommunityName RO
```

```
! 设置可读可写的团体名为 MyPrivateCommunityName  
snmp-server community MyPrivateCommunityName RW
```

对于 Solaris 主机来说，你可以通过 `/etc/snmp/conf/snmpd.conf` 文件来进行定义：

```
read-community MyPublicCommunityName  
write-community MyPrivateCommunityName
```

对这些设备发起的 SNMP 请求必须通过 `MyPublicCommunityName` 团体名来访问只读变量，或者通过 `MyPrivateCommunityName` 团体名来访问可读写变量。换句话说，团体名在这里就可以当成 SNMP 设备的访问密码。这个机制真的非常差劲，不只是因为 SNMP 第一版会在网络中以明文传输团体名，更重要的是它唯一赖以工作的机制就是“不能泄露的机密”。

后来的 SNMP 版本（主要是第三版）为这个协议加入了更好的安全机制。RFC 3414 和 3415 定义了用户安全模型（User Security Model, USM）和基于视图的访问控制模型（View-Based Access Control Model, VACM）。USM 引入了基于加密的验证机制以及信息的加密传输，而 VACM 为 MIB 对象提供了完善的访问控制机制。我们不会对此进行深入介绍，不过你可能会有兴趣自学，因为第三版正在逐渐普及。我建议你读一下 Net-SNMP 软件包里面的 SNMP 第三版教程。如果你对 USM 和 VACM 特别感兴趣，可以参考 NuDesign Technologies 这个 SNMP 供应商在网上发布的简明教程（<http://www.ndt-inc.com/SNMP/HelpFiles/v3ConfigTutorial/v3ConfigTutorial.html>）。

SNMP 实战

现在你已经学习了基本的 SNMP 理论，让我们来动手实践一下吧。之前你已经看到了如何查询一台机器的系统描述信息（还记得之前的漫游吧），现在让我们来看另外两个例子：查询系统的运行时间和 IP 路由表。

到目前为止，你都是从我这里得知 MIB 中的 SNMP 变量名。其实查询 SNMP 信息有两个步骤：

1. 搜索适合的 MIB 文档。如果你查找的信息是设备无关的，那么你可能会在 RFC 1213 找到它。^[注3]如果你需要与设备相关的变量名（比如某个 VoIP 交换机的前面板上的闪光灯的顏色变量），那么你需要联系交换机厂商的客服，获取一份 MIB

注3： RFC 1213 大体上被 RFC 4293、4022 和 4113 代替了。另外 RFC 3418 给 MIB 增加了额外的 SNMPv2 变量。

模块的拷贝。说这段话的时候我非常小心，因为通常会听到有人说“我需要那个设备的 MIB”。要知道全世界只有一个 MIB 结构，其中可以找到所有的变量，而这些设备相关的变量往往在 `private(4)` 分支中。

2. 搜索 MIB 描述信息，直到发现你需要的 SNMP 变量。

为了确保你能完成第二步，^[注4]让我来帮你解释文件格式。

熟悉之后，MIB 描述信息就变得比较容易阅读，它们看上去类似于程序中的变量定义。这并非偶然，因为它们本来就是变量定义。如果设备制造商认真负责的话，你还能在模块文件中读到大量的注释，类似于高质量的源代码文件。

MIB 信息是按照抽象语法表示法1（Abstract Syntax Notation One, ASN.1）的子集来书写的，ASN.1 是开放系统互连（Open Systems Interconnection, OSI）的标准记法。关于这个子集的详细描述可以在管理信息结构（Structure for Management Information, SMI）RFC 系列中找到，这些 RFC 主要是对 SNMP 协议 RFC 和 MIB RFC 做一些补充。例如，目前的 SNMP 协议定义可以参考 RFC 3416，最新的基础 MIB 描述在 RFC 3418 中定义，而这个 MIB 的 SMI 则是由 RFC 2578 定义的。我之所以提到这些 RFC，是因为在搜索某些 SNMP 专题的时候常常需要反复阅读这三个文件。

现在，让我们在通过 SNMP 获取主机的系统运行时间这个任务上应用这些知识。这个 SNMP 变量相当平常，所以它在 RFC 1213 中定义的几率相当高。简单地在 RFC 1213 中搜索“uptime”会得到如下的 ASN.1 片段：

```
sysUpTime OBJECT-TYPE
    SYNTAX  TimeTicks
    ACCESS  read-only
    STATUS  mandatory
    DESCRIPTION
        "The time (in hundredths of a second) since the
        network management portion of the system was last
        re-initialized."
    ::= { system 3 }
```

让我们逐行解释这段定义代码：

`sysUpTime OBJECT-TYPE`

这用来定义一个名叫 `sysUpTime` 的对象。

注4：如果你有一个好的 GUI MIB 浏览器（比如 *mbrowse* 或者 *jmibbrowser*），那么这个任务会更加容易。另外在这个设备上执行 *snmpwalk* 也能帮你快速定位到需要的 MIB 内容。

SYNTAX TimeTicks

这个对象的类型是 `TimeTicks`。对象类型在我刚才提到的 SMI 中指定了。

ACCESS read-only

这个对象对 SNMP 来说是只读的（只能使用 `get-request`），它不能被修改（不能对它执行 `set-request`）。

STATUS mandatory

这个对象对任何 SNMP 代理来说都是必须实现的。

DESCRIPTION...

这是一段关于对象的文本描述。请务必仔细阅读这段文本。比如说在这里我们读到一段有趣的信息：`sysUpTime` 只能显示自“系统的网络管理组件初始化以来”的时间。这意味着这个时间其实是从系统的 SNMP 代理最近一次启动（或者重启）之后才开始计算的。在绝大多数情况下这个时间几乎就是开机时间，但是如果你发现了异常，那么代理的启动时间往往是问题的根源。

::= { system 3 }

这是对象在 MIB 树中的位置。这里说明了 `sysUpTime` 对象定位在系统对象组树的第三个分支。另外，这也是此对象 OID 串中的一段，稍后会用到。

如果我们需要从 `solarisbox` 主机上的只读团体中查询这个变量，可以使用下面的 Net-SNMP 命令：

```
$ snmpget -v 1 -c MyPublicCommunityName solarisbox system.sysUpTime.0
```

这会返回：

```
system.sysUpTime.0 = Timeticks: (5126167) 14:14:21.67
```

这意味着上次代理启动已经是 14 个小时之前的事了。

注意： 这个附录中的范例假定我们的 SNMP 代理已经被配置为允许从指定的主机进行远程查询。通常都应该有这样的设置。

“指定的查询来源”是非常值得实施的安全理念。如果有足够的资源，很有必要对每一台主机上的每一个服务进行这样的限制。对于那些不必要的网络服务，应该把它们关掉。如果必须要开启这个服务，请尽可能把服务的“指定的查询来源”限制在最小范围内。

现在是时候完成第二个更加高级的 SNMP 任务了。这里我们要导出某个设备的 IP 路由表。这个任务之所以复杂，原因是需要处理表状的标量数据集合。我们需要调用 `get-next-request` PDU 来获取数据。任务的第一步仍然是搜索 IP 路由表的 MIB 定义。在 RFC 1213 中搜索“route”，我们应该能得到以下的定义：

```
-- The IP routing table contains an entry for each route
-- presently known to this entity.
ipRouteTable OBJECT-TYPE
    SYNTAX SEQUENCE OF IpRouteEntry
    ACCESS not-accessible
    STATUS mandatory
    DESCRIPTION
        "This entity's IP Routing table."
    ::= { ip 21 }
```

这些定义和之前的定义并没有太大的差别。主要的不同在于 ACCESS 和 SYNTAX 两行。这里的 ACCESS 行有些吓人，不过它的意思其实是说这个对象代表的是一个表，而不是可以直接查询的变量。SYNTAX 这一行的意思是表中包含的是一系列的 IpRouteEntry 对象。让我们再来看看 IpRouteEntry 的定义：

```
ipRouteEntry OBJECT-TYPE
    SYNTAX IpRouteEntry
    ACCESS not-accessible
    STATUS mandatory
    DESCRIPTION
        "A route to a particular destination."
    INDEX { ipRouteDest }
    ::= { ipRouteTable 1 }
```

这里的 ACCESS 行告诉我们这个对象仍然是用来代表中的一行数据。不过这个定义中还有一些另外的线索，它告诉我们每行的数据都可以从一个索引对象来获得，它的名字是 ipRouteDest。

如果这种多层的定义让你不知所云，那么你可以把它和 Perl 对照一下。假设我们正在处理的是一个 Perl 数据结构：列表的哈希。哈希的键是这里的 ipRouteDest 变量，而哈希的值则是那一行路由数据的列表引用。

从 ipRouteEntry 的定义中能看到这些：

```
ipRouteEntry ::=
    SEQUENCE {
        ipRouteDest
            IpAddress,
        ipRouteIfIndex
            INTEGER,
        ipRouteMetric1
            INTEGER,
        ipRouteMetric2
            INTEGER,
        ipRouteMetric3
            INTEGER,
        ipRouteMetric4
            INTEGER,
        ipRouteNextHop
```

```

        IpAddress,
    ipRouteType
        INTEGER,
    ipRouteProto
        INTEGER,
    ipRouteAge
        INTEGER,
    ipRouteMask
        IpAddress,
    ipRouteMetric5
        INTEGER,
    ipRouteInfo
        OBJECT IDENTIFIER
}

```

现在你能看到这个表的每一行的元素了。MIB 紧接着描述每个元素，下面是前两个元素的定义：

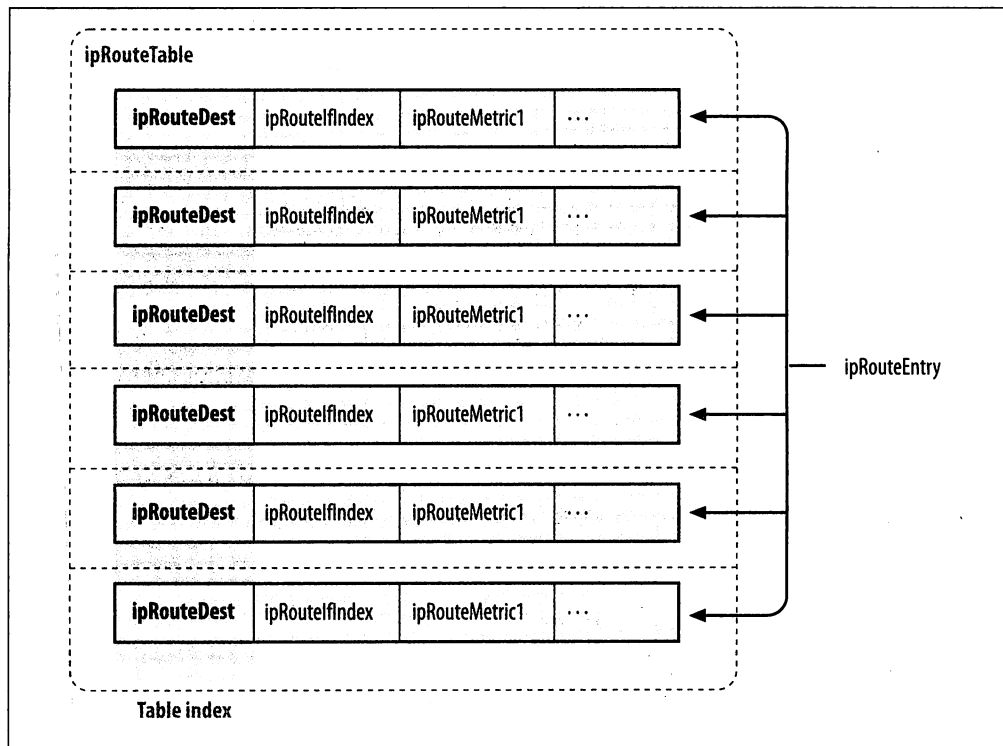
```

ipRouteDest OBJECT-TYPE
    SYNTAX  IpAddress
    ACCESS  read-write
    STATUS  mandatory
    DESCRIPTION
        "The destination IP address of this route. An
        entry with a value of 0.0.0.0 is considered a
        default route. Multiple routes to a single
        destination can appear in the table, but access to
        such multiple entries is dependent on the table-
        access mechanisms defined by the network
        management protocol in use."
    ::= { ipRouteEntry 1 }

ipRouteIfIndex OBJECT-TYPE
    SYNTAX  INTEGER
    ACCESS  read-write
    STATUS  mandatory
    DESCRIPTION
        "The index value which uniquely identifies the
        local interface through which the next hop of this
        route should be reached. The interface identified
        by a particular value of this index is the same
        interface as identified by the same value of
        ifIndex."
    ::= { ipRouteEntry 2 }

```

图G-3展示了以上的 ipRouteTable MIB 结构。



图G-3: ipRouteTable 结构及其索引

一旦你明白了这些 MIB 的结构，下一步就是查询相关的信息了。这一步也可以称为“表遍历 (table traversal)”。大多数的 SNMP 软件包都带有一个叫做 `snmptable` 或者 `snmp-tbl` 的工具，能够完成这个任务，不过它们往往缺乏灵活性。比如，你可能并不希望导出整张路由表，只是需要那些 `ipRouteNextHop` 字段。另外，有些 Perl 的 SNMP 模块并没有对 SNMP 树遍历的支持。考虑到这些因素，我们有必要介绍如何手动遍历。

为了让这个过程更加容易理解，我们先展示期望的结果信息，这有助于我们了解这张表的每一行的内容。如果我们登录到目标主机并且运行 `netstat -nr` 命令来转储 IP 路由表，它的输出应该类似这样：

default	192.168.1.1	UGS	0	215345	tu0
127.0.0.1	127.0.0.1	UH	8	5404381	lo0
192.168.1/24	192.168.1.189	U	15	9222638	tu0

这个输出中还包含了默认的内部回绕接口和本网段的路由。

现在让我们来看看如何使用 Net-SNMP 命令行工具来逐步获得需要的信息。对于这个任

务，我们主要关心的是输出的前两列（路由目标和下一跳的地址）。我们首先尝试获得表中第一行的前两个变量。请注意，下面的命令非常长，必须分行打印才能正常显示：

```
$ snmpgetnext -v 1 -c public computer \  
ip.ipRouteTable.ipRouteEntry.ipRouteDest \  
ip.ipRouteTable.ipRouteEntry.ipRouteNextHop  
ip.ipRouteTable.ipRouteEntry.ipRouteDest.0.0.0.0 = IpAddress: 0.0.0.0  
ip.ipRouteTable.ipRouteEntry.ipRouteNextHop.0.0.0.0 = IpAddress: 192.168.1.1
```

这里需要注意输出中的两个问题：首先是输出的数据，尤其是在等号之前的数据，这里的 0.0.0.0 意味着“默认路由”，所以这行对应着路由表的第一行；其次要注意的是 .0.0.0.0 已经成了变量名的一部分，这是因为这样可以用来作为 ipRouteEntry 条目的索引。

现在我们看到了第一行的数据，可以开始运行第二个 get-next-request 命令了。get-next-request 总是会返回下一个 MIB 元素，所以我们把它放到命令中，用来获取下一行的数据：

```
$ snmpgetnext -v 1 -c public computer \  
ip.ipRouteTable.ipRouteEntry.ipRouteDest.0.0.0.0\  
ip.ipRouteTable.ipRouteEntry.ipRouteNextHop.0.0.0.0  
ip.ipRouteTable.ipRouteEntry.ipRouteDest.127.0.0.1 = IpAddress: 127.0.0.1  
ip.ipRouteTable.ipRouteEntry.ipRouteNextHop.127.0.0.1 = IpAddress: 127.0.0.1
```

现在你应该可以猜到下一步要怎么做了吧。我们可以用 ip.ipRouteTable.ipRouteEntry.ipRouteDest.127.0.0.1 作为 127.0.0.1 的索引，再次调用 get-next-request 命令：

```
$ snmpgetnext -v 1 -c public computer \  
ip.ipRouteTable.ipRouteEntry.ipRouteDest.127.0.0.1 \  
ip.ipRouteTable.ipRouteEntry.ipRouteNextHop.127.0.0.1  
ip.ipRouteTable.ipRouteEntry.ipRouteDest.192.168.1 = IpAddress: 192.168.1.0  
ip.ipRouteTable.ipRouteEntry.ipRouteNextHop.192.168.1.0 = IpAddress: 192.168.1.189
```

对照之前 netstat 命令的输出，你应该发现我们已经完成了任务，转储了所有的 IP 路由表。但是如果我们没有办法引用 netstat 的输出怎么办呢？大多数情况下，你只需要继续执行查询命令就可以了：

```
$ snmpgetnext -v 1 -c public computer \  
ip.ipRouteTable.ipRouteEntry.ipRouteDest.192.168.1.0 \  
ip.ipRouteTable.ipRouteEntry.ipRouteNextHop.192.168.1.0  
ip.ipRouteTable.ipRouteEntry.ipRouteIfIndex.0.0.0.0 = 1  
ip.ipRouteTable.ipRouteEntry.ipRouteType.0.0.0.0 = indirect(4)
```

这里我们没有获得正常的输出，我们要求返回的是 ipRouteDest 和 ipRouteNextHop，但是收到的是 ipRouteIfIndex 和 ipRouteType。这意味着我们已经完全遍历了 ipRouteTable 表。这里 get-next-request 这个 SNMP PDU 已经尽力完成了任务，它返

回了 MIB 表中的“第一个字典顺序的后继者”。回头看看之前摘录自 RFC 1213 的 `ipRouteEntry` 的定义，你就会发现 `ipRouteIfIndex(2)` 紧跟在 `ipRouteDest(1)` 之后，而 `ipRouteType(8)` 紧跟在 `ipRouteNextHop(7)` 之后。

关于“如何知道表遍历结束”这个问题，它的答案是“结束循环程序迭代的条件”。这个条件判断其实也就是对返回字符串的 OID 前缀的检查。比如对于 `ipRouteDest` 查询，它的响应应该包含 `ip.ipRouteTable.ipRouteEntry.ipRouteDest` 或者 `1.3.6.1.2.1.4.21.1.1`。

现在你已经有了最基础的 SNMP 知识，可以回到第 12 章进一步学习如何从 Perl 来使用它。你还可以参考第 12 章末尾的参考资料来获得更多关于 SNMP 的信息。

作者简介

David N. Blank-Edelman 是美国东北大学计算机和信息科学学院的技术总监。他在大型跨平台的系统/网络管理方面有二十五年的经验，其中包括在布兰迪斯大学、剑桥技术集团以及麻省理工学院媒体实验室的历练。他还是 Large Installation System Administration (LISA) 大会 2005 年的主席，也是 2006 年的特约演讲者和联合主席。

封面介绍

《使用Perl实现系统管理自动化》（第二版）封面动物是海獭。北美洲的海獭往往在太平洋沿岸的海藻床礁筑穴，因为那里有很多贝类可供食用。从阿拉斯加到加利福尼亚的海滩都很容易发现海獭。

海獭是灵巧又聪明的哺乳动物，众所周知的是它善于使用工具。在水中漂浮的同时，它们可以把蚌这样的贝类放在自己肚子上，然后用石头来砸碎硬壳。

海獭的习性是群居，它们往往喜欢挤在一起，浮在水面上，拼成一个“大木筏”。它们的游泳本领非常高，蹼状脚爪使得它们能在水中自由穿行。厚厚的皮毛能使它们在水中保持干燥。不过这身“皮大衣”也导致它们被大量捕杀，从而危及自己的生存。

O'Reilly Media, Inc. 介绍

O'Reilly Media, Inc. 是世界上在 UNIX、X、Internet 和其他开放系统图书领域具有领导地位的出版公司，同时是联机出版的先锋。

从最畅销的《The Whole Internet User's Guide & Catalog》(被纽约公共图书馆评为二十世纪最重要的 50 本书之一) 到 GNN (最早的 Internet 门户和商业网站)，再到 WebSite (第一个桌面 PC 的 Web 服务器软件)，O'Reilly Media, Inc. 一直处于 Internet 发展的最前沿。

许多书店的反馈表明，O'Reilly Media, Inc. 是最稳定的计算机图书出版商——每一本书都一版再版。与大多数计算机图书出版商相比，O'Reilly Media, Inc. 具有深厚的计算机专业背景，这使得 O'Reilly Media, Inc. 形成了一个非常不同于其他出版商的出版方针。O'Reilly Media, Inc. 所有的编辑人员以前都是程序员，或者是顶尖级的技术专家。O'Reilly Media, Inc. 还有许多固定的作者群体——他们本身是相关领域的技术专家、咨询专家，而现在编写著作，O'Reilly Media, Inc. 依靠他们及时地推出图书。因为 O'Reilly Media, Inc. 紧密地与计算机业界联系着，所以 O'Reilly Media, Inc. 知道市场上真正需要什么图书。